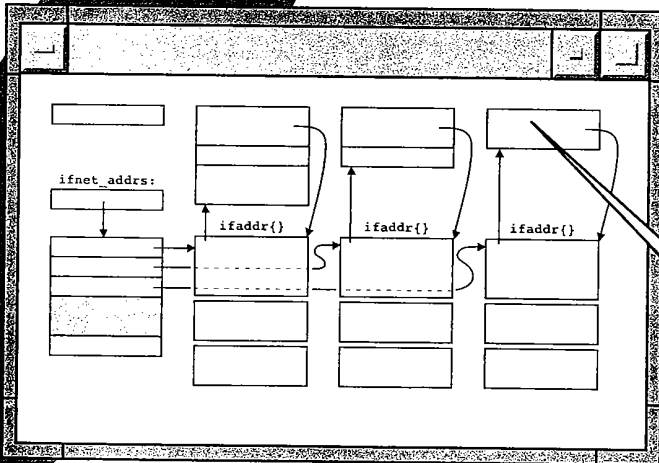


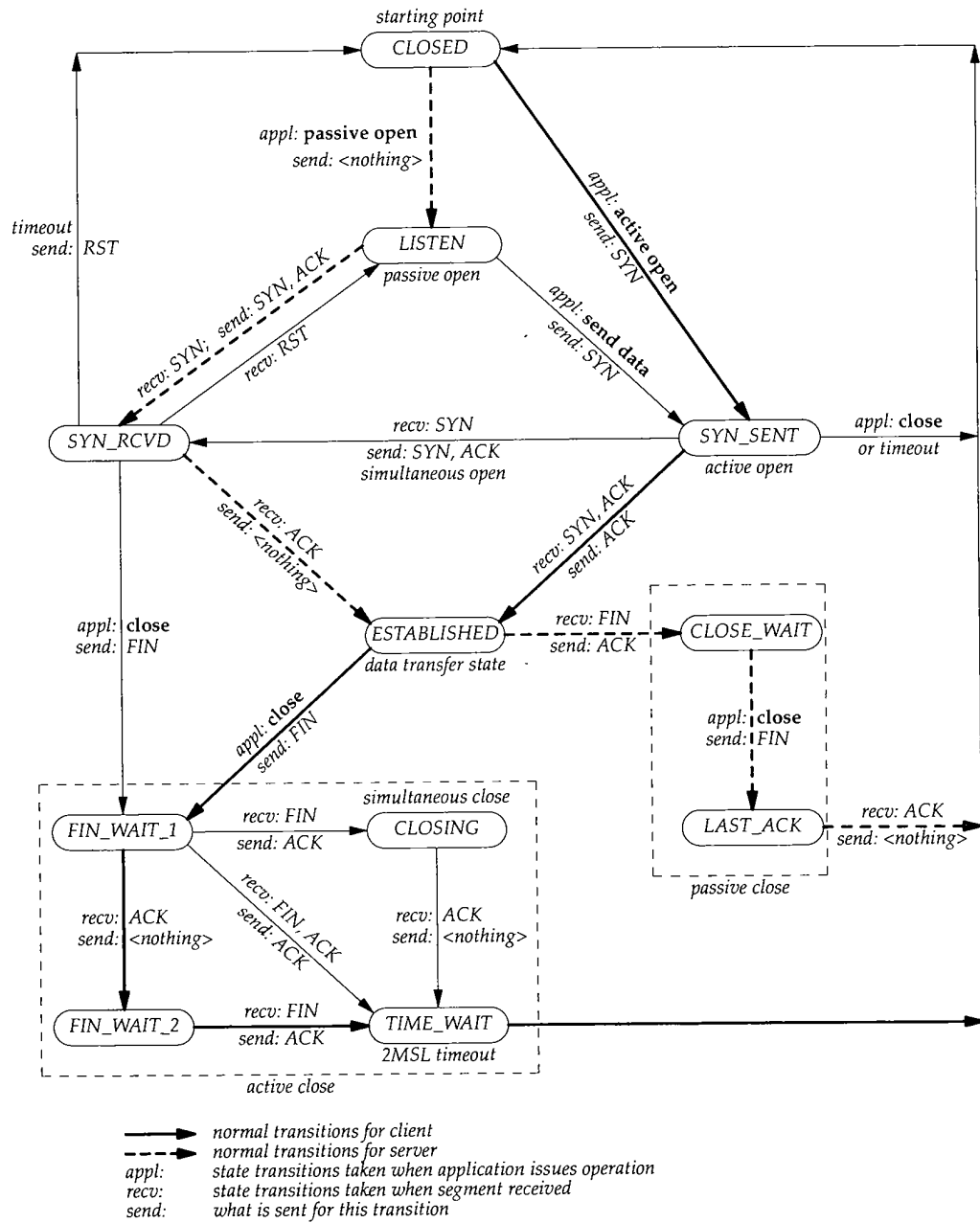
# TCP/IP Illustrated Volume 1

The Implementation

Gary R. Wright  
With Richard Stevens



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



TCP state transition diagram.

## Structure Definitions

arpcom	80	mrt	419
arphdr	682	mrtctl	420
		msghdr	482
bpf_d	1033		
bpf_hdr	1029	osockaddr	75
bpf_if	1029		
		pdevinit	78
cmsghdr	482	protosw	188
domain	187	radix_mask	578
		radix_node	575
ether_arp	682	radix_node_head	574
ether_header	102	rawcb	647
ether_multi	342	route	220
		route_cb	625
icmp	308	rt_addrinfo	623
ifaddr	73	rtentry	579
ifa_msghdr	622	rt_metrics	580
ifconf	117	rt_msghdr	622
if_msghdr	622		
ifnet	67	selinfo	531
ifqueue	71	sl_softc	83
ifreq	117	sockaddr	75
igmp	384	sockaddr_dl	87
in_addr	160	sockaddr_in	160
in_aliasreq	174	sockaddr_inarp	701
in_ifaddr	161	sockbuf	476
in_multi	345	socket	438
inpcb	716	socket_args	444
iovec	481	sockproto	626
ip	211	sysent	443
ipasfrag	287		
ip_moptions	347	tcpcb	804
ip_mreq	356	tcp_debug	916
ipoption	265	tcphdr	801
ipovly	760	tcpiphdr	803
ipq	286	timeval	106
ip_srcrt	258		
ip_timestamp	262	udphdr	759
		udpiphdr	759
le_softc	80	uio	485
lgrplctl	411		
linger	542	vif	406
llinfo_arp	682	vifctl	407
mbuf	38	walkarg	632



## Praise for *TCP/IP Illustrated, Volume 1: The Protocols*

---

"*TCP/IP Illustrated* has already become my most-likely-to-have-the-answer reference book, the first resource I turn to with a networking question. The book is, all publisher hype aside, an instant classic, and I, for one, am thrilled that something like this is now available."

— Vern Paxson, *login:*, March/April 1994

"This is sure to be the bible for TCP/IP developers and users."

— Robert A. Ciampa, Network Engineer, Synernetics, division of 3COM

"... the difference is that Stevens wants to show as well as tell about the protocols. His principal teaching tools are straight-forward explanations, exercises at the ends of chapters, byte-by-byte diagrams of headers and the like, and listings of actual traffic as examples."

— Walter Zintz, *Unix World*, December 1993

"*TCP/IP Illustrated, Volume 1* is based on practical examples that reinforce the theory — distinguishing this book from others on the subject, and making it both readable and informative."

— Peter M. Haverlock, Consultant, IBM TCP/IP Development

"While all of Stevens' books are excellent, this new opus is awesome. Although many books describe the TCP/IP protocols, the author provides a level of depth and real-world detail lacking from the competition."

— Steven Baker, *Unix Review*, March 1994

"*TCP/IP Illustrated, Volume 1* is an excellent reference for developers, network administrators or anyone who needs to understand TCP/IP technology."

— Bob Williams, V.P. Marketing, NetManage, Inc.

"W. Richard Stevens has produced a fine text and reference work."

— Scott Bradner, Consultant, Harvard University OIT/NSD

"Even marketing weenies (of a technical bent) will appreciate this book, as it is clearly written, and uses lots of diagrams. I especially like the author's thoughtful use of asides—set in smaller type and indented—to explain this or that concept. "

— Ron Jeffries, *ATM USER*, January 1994

"Stevens takes a subject that has been written about rather prolifically, TCP/IP, and does something fresh and useful with it."

— Jason Levitt, *Open Systems Today*, March 7, 1994

## More Praise for *TCP/IP Illustrated, Volume 1: The Protocols*

---

"This book is a stone jewel. ... Written by W. Richard Stevens, this book probably provides the most comprehensive view of TCP/IP available today in print."

— *Boardwatch*, April/May 1994

"...you can't get a better understanding of the workings of TCP/IP anywhere."

— Tom Nolle, *Netwatcher*, January 1994

"The book covers all the basic TCP/IP applications, including Telnet, NFS (Network File System), FTP (file transfer protocol) and TFTP (trivial FTP)."

— *Data Communications*, January 21, 1994

"The diagrams he uses are excellent and his writing style is clear and readable. Please read it and keep it on your bookshelf."

— Elizabeth Zinkann, *Sys Admin*, November 1993

"Stevens' Unix-oriented investigations will be invaluable to the network programmer or specialist who wishes to really understand how the TCP/IP stack is put together."

— Joel Snyder, *Internet World*, March/April 1994 issue

"All aspects of the transmission control protocol/Internet protocol (TCP/IP) are covered here, from link layer and static/dynamic routing implementations to applications such as SNMP and Telnet."

— *Telecommunications*, March 1994

"The author of *TCP/IP Illustrated* has succeeded in creating another indispensable tome of networking knowledge. This is the most comprehensible and complete book I have read on TCP/IP. It takes a different slant than other books, by presenting not only details of TCP, IP, ARP, ICMP, routing, etc., but actually shows these protocols (and common Internet tools) in action."

— Eli Charne, *ConneXions*, July 1994

"The word 'illustrated' distinguishes this book from its many rivals."

— Stan Kelly-Bootle, *Unix Review*, December 1993

TCP/IP Illustrated, Volume 2

---

## Addison-Wesley Professional Computing Series

---

Brian W. Kernighan, Consulting Editor

- Matthew H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*  
David R. Butenhof, *Programming with POSIX® Threads*  
Brent Callaghan, *NFS Illustrated*  
Tom Cargill, *C++ Programming Style*  
William R. Cheswick/Steven M. Bellovin/Aviel D. Rubin, *Firewalls and Internet Security, Second Edition: Repelling the Wily Hacker*  
David A. Curry, *UNIX® System Security: A Guide for Users and System Administrators*  
Stephen C. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*  
Dan Farmer/Wietse Venema, *Forensic Discovery*  
Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*  
Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns CD: Elements of Reusable Object-Oriented Software*  
Peter Hagggar, *Practical Java™ Programming Language Guide*  
David R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*  
Mark Harrison/Michael McLennan, *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*  
Michi Henning/Steve Vinoski, *Advanced CORBA® Programming with C++*  
Brian W. Kernighan/Rob Pike, *The Practice of Programming*  
S. Keshav, *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*  
John Lakos, *Large-Scale C++ Software Design*  
Scott Meyers, *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*  
Scott Meyers, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*  
Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*  
Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*  
Robert B. Murray, *C++ Strategies and Tactics*  
David R. Musser/Gillmer J. Derge/Atul Saini, *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*  
John K. Ousterhout, *Tcl and the Tk Toolkit*  
Craig Partridge, *Gigabit Networking*  
Radia Perlman, *Interconnections, Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*  
Stephen A. Rago, *UNIX® System V Network Programming*  
Eric S. Raymond, *The Art of UNIX Programming*  
Marc J. Rochkind, *Advanced UNIX Programming, Second Edition*  
Curt Schimmel, *UNIX® Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*  
W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*  
W. Richard Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX® Domain Protocols*  
W. Richard Stevens/Bill Fenner/Andrew M. Rudoff, *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*  
W. Richard Stevens/Stephen A. Rago, *Advanced Programming in the UNIX® Environment, Second Edition*  
W. Richard Stevens/Gary R. Wright, *TCP/IP Illustrated Volumes 1-3 Boxed Set*  
John Viega/Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*  
Gary R. Wright/W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*  
Ruixi Yuan/W. Timothy Strayer, *Virtual Private Networks: Technologies and Solutions*

Visit [www.awprofessional.com/series/professionalcomputing](http://www.awprofessional.com/series/professionalcomputing) for more information about these titles.



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

---

**Gary R. Wright**  
**W. Richard Stevens**



**ADDISON-WESLEY**

---

Boston • San Francisco • New York • Toronto • Montreal  
London • Munich • Paris • Madrid  
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this 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. They 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  
201 W. 103rd Street  
Indianapolis, IN 46290  
(800) 428-5331  
corpsales@pearsoned.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 indexes.

Contents: v. 1. The protocols -- v.2. The implementation

I. TCP/IP (Computer network protocol) I Wright,  
Gary R., II. Title. III. Series.  
TK5105.55.S74 1994 004.6'2 93-40000  
ISBN 0-201-63346-9 (v.1)  
ISBN 0-201-63354-X (v.2)

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

Copyright © 1995 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or other-wise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

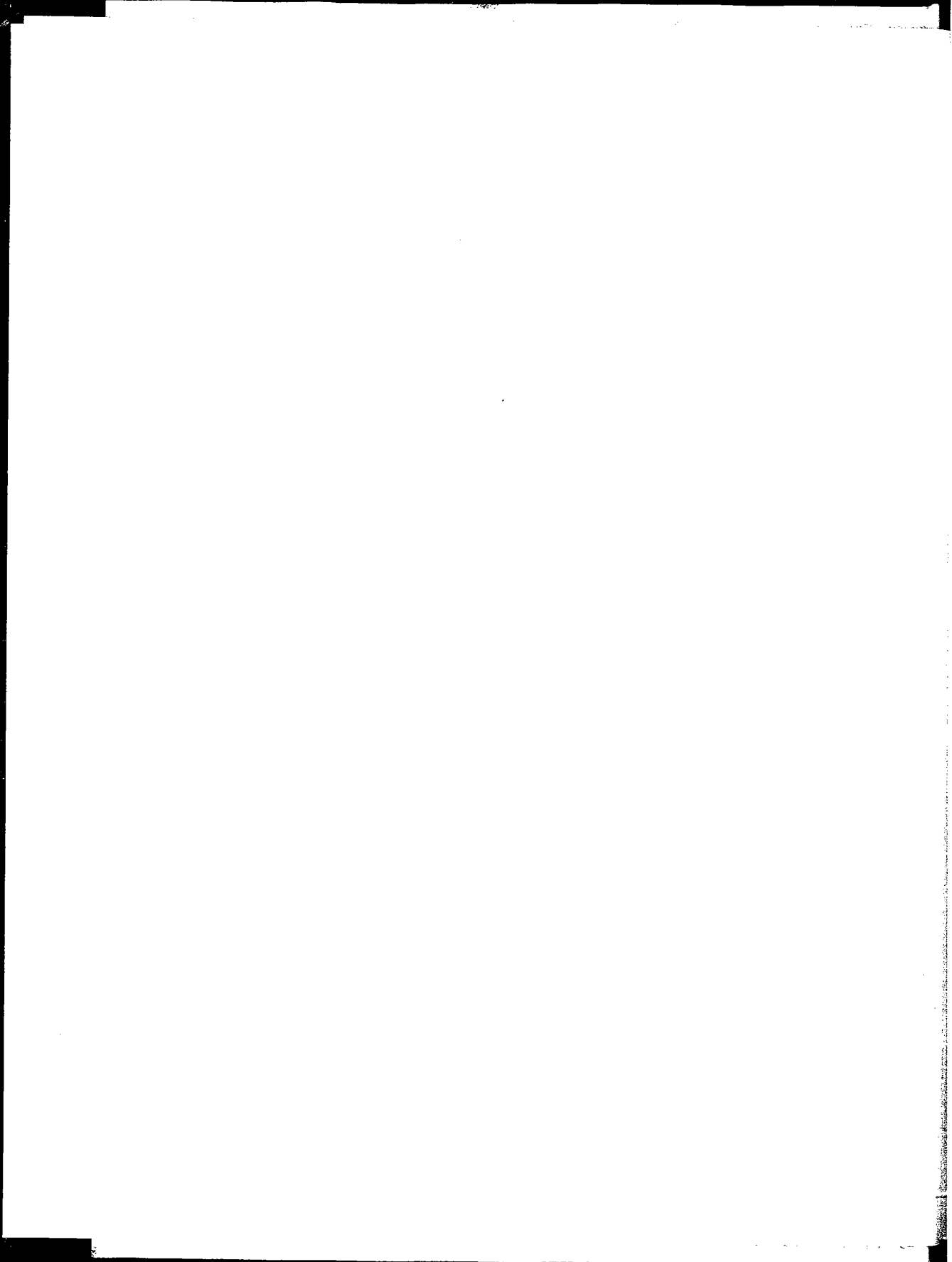
ISBN 0-201-63354-X

Text printed in the United States on recycled paper at Courier Westford in Westford, Massachusetts.

24th Printing September 2010

*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.



# Contents

<b>Preface</b>		<b>xix</b>
<b>Chapter 1. Introduction</b>		<b>1</b>
1.1	Introduction	1
1.2	Source Code Presentation	1
1.3	History	3
1.4	Application Programming Interfaces	5
1.5	Example Program	5
1.6	System Calls and Library Functions	7
1.7	Network Implementation Overview	9
1.8	Descriptors	10
1.9	Mbufs (Memory Buffers) and Output Processing	15
1.10	Input Processing	19
1.11	Network Implementation Overview Revisited	22
1.12	Interrupt Levels and Concurrency	23
1.13	Source Code Organization	26
1.14	Test Network	28
1.15	Summary	29
<b>Chapter 2. Mbufs: Memory Buffers</b>		<b>31</b>
2.1	Introduction	31
2.2	Code Introduction	36
2.3	Mbuf Definitions	37
2.4	mbuf Structure	38
2.5	Simple Mbuf Macros and Functions	40
2.6	m_devget and m_pullup Functions	44

2.7	Summary of Mbuf Macros and Functions	51	
2.8	Summary of Net/3 Networking Data Structures	54	
2.9	m_copy and Cluster Reference Counts	56	
2.10	Alternatives	60	
2.11	Summary	60	
<b>Chapter 3.</b>	<b>Interface Layer</b>		<b>63</b>
3.1	Introduction	63	
3.2	Code Introduction	64	
3.3	ifnet Structure	65	
3.4	ifaddr Structure	73	
3.5	sockaddr Structure	74	
3.6	ifnet and ifaddr Specialization	76	
3.7	Network Initialization Overview	77	
3.8	Ethernet Initialization	80	
3.9	SLIP Initialization	82	
3.10	Loopback Initialization	85	
3.11	if_attach Function	85	
3.12	ifinit Function	93	
3.13	Summary	94	
<b>Chapter 4.</b>	<b>Interfaces: Ethernet</b>		<b>95</b>
4.1	Introduction	95	
4.2	Code Introduction	96	
4.3	Ethernet Interface	98	
4.4	ioctl System Call	114	
4.5	Summary	125	
<b>Chapter 5.</b>	<b>Interfaces: SLIP and Loopback</b>		<b>127</b>
5.1	Introduction	127	
5.2	Code Introduction	127	
5.3	SLIP Interface	128	
5.4	Loopback Interface	150	
5.5	Summary	153	
<b>Chapter 6.</b>	<b>IP Addressing</b>		<b>155</b>
6.1	Introduction	155	
6.2	Code Introduction	158	
6.3	Interface and Address Summary	158	
6.4	sockaddr_in Structure	160	
6.5	in_ifaddr Structure	161	
6.6	Address Assignment	161	
6.7	Interface ioctl Processing	177	
6.8	Internet Utility Functions	181	
6.9	ifnet Utility Functions	182	
6.10	Summary	183	

<b>Chapter 7.</b>	<b>Domains and Protocols</b>	<b>185</b>
7.1	Introduction	185
7.2	Code Introduction	186
7.3	domain Structure	187
7.4	protosw Structure	188
7.5	IP domain and protosw Structures	191
7.6	pfindproto and pfindtype Functions	196
7.7	pftlinput Function	198
7.8	IP Initialization	199
7.9	sysctl System Call	201
7.10	Summary	204
<b>Chapter 8.</b>	<b>IP: Internet Protocol</b>	<b>205</b>
8.1	Introduction	205
8.2	Code Introduction	206
8.3	IP Packets	210
8.4	Input Processing: ipintr Function	212
8.5	Forwarding: ip_forward Function	220
8.6	Output Processing: ip_output Function	228
8.7	Internet Checksum: in_cksum Function	234
8.8	setsockopt and getsockopt System Calls	239
8.9	ip_sysctl Function	244
8.10	Summary	245
<b>Chapter 9.</b>	<b>IP Option Processing</b>	<b>247</b>
9.1	Introduction	247
9.2	Code Introduction	247
9.3	Option Format	248
9.4	ip_dooptions Function	249
9.5	Record Route Option	252
9.6	Source and Record Route Options	254
9.7	Timestamp Option	261
9.8	ip_insetoptions Function	265
9.9	ip_pchopts Function	269
9.10	Limitations	272
9.11	Summary	272
<b>Chapter 10.</b>	<b>IP Fragmentation and Reassembly</b>	<b>275</b>
10.1	Introduction	275
10.2	Code Introduction	277
10.3	Fragmentation	278
10.4	ip_optcopy Function	282
10.5	Reassembly	283
10.6	ip_reass Function	286
10.7	ip_slowtimo Function	298
10.8	Summary	300

<b>Chapter 11.</b>	<b>ICMP: Internet Control Message Protocol</b>	<b>301</b>
11.1	Introduction	301
11.2	Code Introduction	305
11.3	icmp Structure	308
11.4	ICMP protosw Structure	309
11.5	Input Processing: icmp_input Function	310
11.6	Error Processing	313
11.7	Request Processing	316
11.8	Redirect Processing	321
11.9	Reply Processing	323
11.10	Output Processing	324
11.11	icmp_error Function	324
11.12	icmp_reflect Function	328
11.13	icmp_send Function	333
11.14	icmp_sysctl Function	334
11.15	Summary	335
<b>Chapter 12.</b>	<b>IP Multicasting</b>	<b>337</b>
12.1	Introduction	337
12.2	Code Introduction	340
12.3	Ethernet Multicast Addresses	341
12.4	ether_multi Structure	342
12.5	Ethernet Multicast Reception	344
12.6	in_multi Structure	345
12.7	ip_moptions Structure	347
12.8	Multicast Socket Options	348
12.9	Multicast TTL Values	348
12.10	ip_setmoptions Function	351
12.11	Joining an IP Multicast Group	355
12.12	Leaving an IP Multicast Group	366
12.13	ip_getmoptions Function	371
12.14	Multicast Input Processing: ipintr Function	373
12.15	Multicast Output Processing: ip_output Function	375
12.16	Performance Considerations	379
12.17	Summary	379
<b>Chapter 13.</b>	<b>IGMP: Internet Group Management Protocol</b>	<b>381</b>
13.1	Introduction	381
13.2	Code Introduction	382
13.3	igmp Structure	384
13.4	IGMP protosw Structure	384
13.5	Joining a Group: igmp_joingroup Function	386
13.6	igmp_fasttimo Function	387
13.7	Input Processing: igmp_input Function	391
13.8	Leaving a Group: igmp_leavegroup Function	395
13.9	Summary	396



<b>Chapter 14.</b>	<b>IP Multicast Routing</b>	<b>397</b>
14.1	Introduction	397
14.2	Code Introduction	398
14.3	Multicast Output Processing Revisited	399
14.4	mrouterd Daemon	401
14.5	Virtual Interfaces	404
14.6	IGMP Revisited	411
14.7	Multicast Routing	416
14.8	Multicast Forwarding: ip_mforward Function	424
14.9	Cleanup: ip_router_done Function	433
14.10	Summary	434
<b>Chapter 15.</b>	<b>Socket Layer</b>	<b>435</b>
15.1	Introduction	435
15.2	Code Introduction	436
15.3	socket Structure	437
15.4	System Calls	441
15.5	Processes, Descriptors, and Sockets	445
15.6	socket System Call	447
15.7	getsock and sockargs Functions	451
15.8	bind System Call	453
15.9	listen System Call	455
15.10	tsleep and wakeup Functions	456
15.11	accept System Call	457
15.12	sonewconn and soisconnected Functions	461
15.13	connect System call	464
15.14	shutdown System Call	468
15.15	close System Call	471
15.16	Summary	474
<b>Chapter 16.</b>	<b>Socket I/O</b>	<b>475</b>
16.1	Introduction	475
16.2	Code Introduction	475
16.3	Socket Buffers	476
16.4	write, writev, sendto, and sendmsg System Calls	480
16.5	sendmsg System Call	483
16.6	sendit Function	485
16.7	soSEND Function	489
16.8	read, readv, recvfrom, and recvmsg System Calls	500
16.9	recvmsg System Call	501
16.10	recvit Function	503
16.11	soreceive Function	505
16.12	soreceive Code	510
16.13	select System Call	524
16.14	Summary	534

<b>Chapter 17.</b>	<b>Socket Options</b>	<b>537</b>
17.1	Introduction	537
17.2	Code Introduction	538
17.3	setsockopt System Call	539
17.4	getsockopt System Call	545
17.5	fcntl and ioctl System Calls	548
17.6	getsockname System Call	554
17.7	getpeername System Call	554
17.8	Summary	557
<b>Chapter 18.</b>	<b>Radix Tree Routing Tables</b>	<b>559</b>
18.1	Introduction	559
18.2	Routing Table Structure	560
18.3	Routing Sockets	569
18.4	Code Introduction	570
18.5	Radix Node Data Structures	573
18.6	Routing Structures	578
18.7	Initialization: route_init and rtable_init Functions	581
18.8	Initialization: rn_init and rn_inithead Functions	584
18.9	Duplicate Keys and Mask Lists	587
18.10	rn_match Function	591
18.11	rn_search Function	599
18.12	Summary	599
<b>Chapter 19.</b>	<b>Routing Requests and Routing Messages</b>	<b>601</b>
19.1	Introduction	601
19.2	rtalloc and rtallocl Functions	601
19.3	RTFREE Macro and rtfree Function	604
19.4	rtrequest Function	607
19.5	rt_setgate Function	612
19.6	rtinit Function	615
19.7	rtredirect Function	617
19.8	Routing Message Structures	621
19.9	rt_missmsg Function	625
19.10	rt_ifmsg Function	627
19.11	rt_newaddrmsg Function	628
19.12	rt_msg1 Function	630
19.13	rt_msg2 Function	632
19.14	sysctl_rtable Function	635
19.15	sysctl_dumpentry Function	640
19.16	sysctl_iflist Function	642
19.17	Summary	644
<b>Chapter 20.</b>	<b>Routing Sockets</b>	<b>645</b>
20.1	Introduction	645
20.2	routedomain and protosw Structures	646
20.3	Routing Control Blocks	647

20.4	raw_init Function	647	
20.5	route_output Function	648	
20.6	rt_xaddrs Function	660	
20.7	rt_setmetrics Function	661	
20.8	raw_input Function	662	
20.9	route_usrreq Function	664	
20.10	raw_usrreq Function	666	
20.11	raw_attach, raw_detach, and raw_disconnect Functions	671	
20.12	Summary	672	
<b>Chapter 21.</b>	<b>ARP: Address Resolution Protocol</b>		<b>675</b>
21.1	Introduction	675	
21.2	ARP and the Routing Table	675	
21.3	Code Introduction	678	
21.4	ARP Structures	681	
21.5	arpwhoas Function	683	
21.6	arprequest Function	684	
21.7	arpintr Function	687	
21.8	in_arpinput Function	688	
21.9	ARP Timer Functions	694	
21.10	arpresolve Function	696	
21.11	arplookup Function	701	
21.12	Proxy ARP	703	
21.13	arp_rtrequest Function	704	
21.14	ARP and Multicasting	710	
21.15	Summary	711	
<b>Chapter 22.</b>	<b>Protocol Control Blocks</b>		<b>713</b>
22.1	Introduction	713	
22.2	Code Introduction	715	
22.3	inpcb Structure	716	
22.4	in_pcballoc and in_pcbdetach Functions	717	
22.5	Binding, Connecting, and Demultiplexing	719	
22.6	in_pcblookup Function	724	
22.7	in_pcbbind Function	728	
22.8	in_pcbconnect Function	735	
22.9	in_pcbdisconnect Function	741	
22.10	in_setsockaddr and in_setpeeraddr Functions	741	
22.11	in_pcbnotify, in_rtchange, and in_losing Functions	742	
22.12	Implementation Refinements	750	
22.13	Summary	751	
<b>Chapter 23.</b>	<b>UDP: User Datagram Protocol</b>		<b>755</b>
23.1	Introduction	755	
23.2	Code Introduction	755	
23.3	UDP protosw Structure	758	

23.4	UDP Header	759	
23.5	udp_init Function	760	
23.6	udp_output Function	760	
23.7	udp_input Function	769	
23.8	udp_saveopt Function	781	
23.9	udp_ctlinput Function	782	
23.10	udp_usrreq Function	784	
23.11	udp_sysctl Function	790	
23.12	Implementation Refinements	791	
23.13	Summary	793	
<b>Chapter 24.</b>	<b>TCP: Transmission Control Protocol</b>		<b>795</b>
24.1	Introduction	795	
24.2	Code Introduction	795	
24.3	TCP protosw Structure	801	
24.4	TCP Header	801	
24.5	TCP Control Block	803	
24.6	TCP State Transition Diagram	805	
24.7	TCP Sequence Numbers	807	
24.8	tcp_init Function	812	
24.9	Summary	815	
<b>Chapter 25.</b>	<b>TCP Timers</b>		<b>817</b>
25.1	Introduction	817	
25.2	Code Introduction	819	
25.3	tcp_canceltimers Function	821	
25.4	tcp_fasttimo Function	821	
25.5	tcp_slowtimo Function	822	
25.6	tcp_timers Function	824	
25.7	Retransmission Timer Calculations	831	
25.8	tcp_newtcpcb Function	833	
25.9	tcp_setpersist Function	835	
25.10	tcp_xmit_timer Function	836	
25.11	Retransmission Timeout: tcp_timers Function	841	
25.12	An RTT Example	846	
25.13	Summary	848	
<b>Chapter 26.</b>	<b>TCP Output</b>		<b>851</b>
26.1	Introduction	851	
26.2	tcp_output Overview	852	
26.3	Determine if a Segment Should be Sent	852	
26.4	TCP Options	864	
26.5	Window Scale Option	866	
26.6	Timestamp Option	866	
26.7	Send a Segment	871	
26.8	tcp_template Function	884	
26.9	tcp_respond Function	885	
26.10	Summary	888	

<b>Chapter 27. TCP Functions</b>	<b>891</b>
27.1 Introduction	891
27.2 tcp_drain Function	892
27.3 tcp_drop Function	892
27.4 tcp_close Function	893
27.5 tcp_mss Function	897
27.6 tcp_ctlinput Function	904
27.7 tcp_notify Function	904
27.8 tcp_quench Function	906
27.9 TCP_REASS Macro and tcp_reass Function	906
27.10 tcp_trace Function	916
27.11 Summary	920
<b>Chapter 28. TCP Input</b>	<b>923</b>
28.1 Introduction	923
28.2 Preliminary Processing	925
28.3 tcp_dooptions Function	933
28.4 Header Prediction	934
28.5 TCP Input: Slow Path Processing	941
28.6 Initiation of Passive Open, Completion of Active Open	942
28.7 PAWS: Protection Against Wrapped Sequence Numbers	951
28.8 Trim Segment so Data is Within Window	954
28.9 Self-Connects and Simultaneous Opens	960
28.10 Record Timestamp	963
28.11 RST Processing	963
28.12 Summary	965
<b>Chapter 29. TCP Input (Continued)</b>	<b>967</b>
29.1 Introduction	967
29.2 ACK Processing Overview	967
29.3 Completion of Passive Opens and Simultaneous Opens	967
29.4 Fast Retransmit and Fast Recovery Algorithms	970
29.5 ACK Processing	974
29.6 Update Window Information	981
29.7 Urgent Mode Processing	983
29.8 tcp_pulloutofband Function	986
29.9 Processing of Received Data	988
29.10 FIN Processing	990
29.11 Final Processing	992
29.12 Implementation Refinements	994
29.13 Header Compression	995
29.14 Summary	1004
<b>Chapter 30. TCP User Requests</b>	<b>1007</b>
30.1 Introduction	1007
30.2 tcp_usrreq Function	1007
30.3 tcp_attach Function	1018
30.4 tcp_disconnect Function	1019

30.5	tcp_usrclosed Function	1021	
30.6	tcp_ctloutput Function	1022	
30.7	Summary	1025	
<b>Chapter 31.</b>	<b>BPF: BSD Packet Filter</b>		<b>1027</b>
31.1	Introduction	1027	
31.2	Code Introduction	1028	
31.3	bpf_if Structure	1029	
31.4	bpf_d Structure	1032	
31.5	BPF Input	1040	
31.6	BPF Output	1046	
31.7	Summary	1047	
<b>Chapter 32.</b>	<b>Raw IP</b>		<b>1049</b>
32.1	Introduction	1049	
32.2	Code Introduction	1050	
32.3	Raw IP protosw Structure	1051	
32.4	rip_init Function	1053	
32.5	rip_input Function	1053	
32.6	rip_output Function	1056	
32.7	rip_usrreq Function	1058	
32.8	rip_ctloutput Function	1063	
32.9	Summary	1065	
<b>Epilogue</b>			<b>1067</b>
<b>Appendix A.</b>	<b>Solutions to Selected Exercises</b>		<b>1069</b>
<b>Appendix B.</b>	<b>Source Code Availability</b>		<b>1093</b>
<b>Appendix C.</b>	<b>RFC 1122 Compliance</b>		<b>1097</b>
C.1	Link-Layer Requirements	1097	
C.2	IP Requirements	1098	
C.3	IP Options Requirements	1102	
C.4	IP Fragmentation and Reassembly Requirements	1104	
C.5	ICMP Requirements	1105	
C.6	Multicasting Requirements	1110	
C.7	IGMP Requirements	1111	
C.8	Routing Requirements	1111	
C.9	ARP Requirements	1113	
C.10	UDP Requirements	1113	
C.11	TCP Requirements	1115	
<b>Bibliography</b>			<b>1125</b>
<b>Index</b>			<b>1133</b>

# Preface

## Introduction

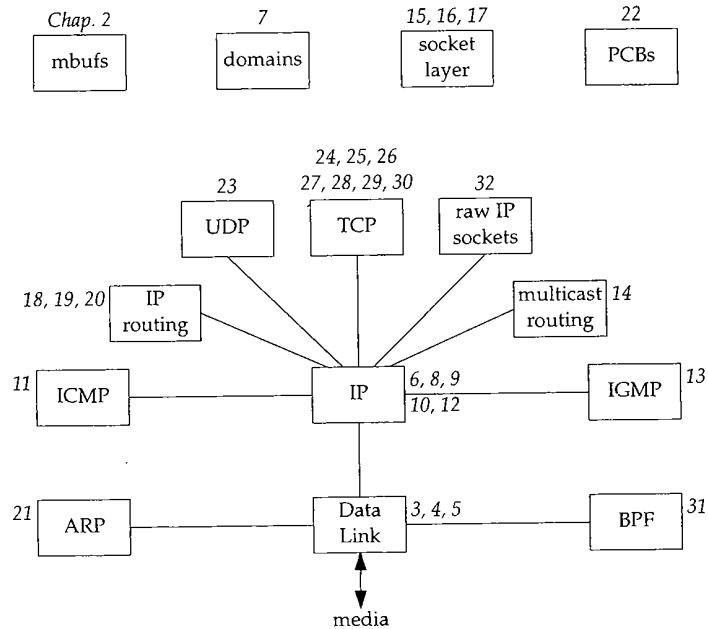
This book describes and presents the source code for the common reference implementation of TCP/IP: the implementation from the Computer Systems Research Group (CSRG) at the University of California at Berkeley. Historically this has been distributed with the 4.x BSD system (Berkeley Software Distribution). 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 (p. 4) 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.

## 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.



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).

## 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 text—there 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.

### Source Code Copyright

All of the source code presented in this book, other than Figures 1.2 and 8.27, is from the 4.4BSD-Lite distribution. This software is publicly available through many sources (Appendix B).

All of this source code contains the following copyright notice.

```
/*
 * Copyright (c) 1982, 1986, 1988, 1990, 1993, 1994
 *   The Regents of the University of California. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the 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.
 * 3. All advertising materials mentioning features or use of this software
 * must display the following acknowledgement:
 *   This product includes software developed by the University of
 *   California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 * may be used to endorse or promote products derived from this software
 * without specific prior written permission.
 *
 * 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.
 */
```

## 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, Chakravarthi 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  
*Middletown, Connecticut*  
November 1994

W. Richard Stevens  
*Tucson, Arizona*  
[www.kohala.com](http://www.kohala.com)

# *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.

## **1.2 Source Code Presentation**

Presenting 15,000 lines of source code, regardless of the topic, is a challenge in itself. The following format is used for all the source code in the text:

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

### Set congestion window to one segment

387-388 This is the `tcp_quench` function from the file `tcp_subr.c`. These source file-names refer to files in the 4.4BSD-Lite distribution, which we describe in Section 1.13. Each nonblank line is numbered. The text describing portions of the code begins with the starting and ending line numbers in the left margin, as shown with this paragraph. Sometimes the paragraph is preceded by a short descriptive heading, providing a summary statement of the code being described.

The source code has been left as is from the 4.4BSD-Lite distribution, including occasional bugs, which we note and discuss when encountered, and occasional editorial comments from the original authors. The code has been run through the GNU Indent program to provide consistency in appearance. The tab stops have been set to four-column boundaries to allow the lines to fit on a page. Some `#ifdef` statements and their corresponding `#endif` have been removed when the constant is always defined (e.g., `GATEWAY` and `MROUTING`, since we assume the system is operating as a router and as a multicast router). All `register` specifiers have been removed. Sometimes a comment has been added and typographical errors in the comments have been fixed, but otherwise the code has been left alone.

The functions vary in size from a few lines (`tcp_quench` shown earlier) to `tcp_input`, which is the biggest at 1100 lines. Functions that exceed about 40 lines are normally broken into pieces, which are shown one after the other. Every attempt is made to place the code and its accompanying description on the same page or on facing pages, but this isn't always possible without wasting a large amount of paper.

Many cross-references are provided to other functions that are described in the text. To avoid appending both a figure number and a page number to each reference, 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. Since the source code in the book is taken from the publicly available 4.4BSD-Lite release, you can easily obtain a copy: Appendix B details various ways. Sometimes it helps to have an on-line copy to search through [e.g., with the Unix `grep(1)` program] as you follow the text.

Each chapter that describes a source code module normally begins with a listing of the source files being described, followed by the global variables, the relevant statistics maintained by the code, some sample statistics from an actual system, and finally the SNMP variables related to the protocol being described. The global variables are often

defined across various source files and headers, so we collect them in one table for easy reference. Showing all the statistics at this point simplifies the later discussion of the code when the statistics are updated. Chapter 25 of Volume 1 provides all the details on SNMP. Our interest in this text is in the information maintained by the TCP/IP routines in the kernel to support an SNMP agent running on the system.

### Typographical Conventions

In the figures throughout the text we use a constant-width font for variable names and the names of structure members (`m_next`), a slanted constant-width font for names that are defined constants (`NULL`) or constant values (`512`), and a bold constant-width font with braces for structure names (`mbuf { }`). Here is an example:

<code><b>mbuf { }</b></code>	
<code>m_next</code>	<i>NULL</i>
<code>m_len</code>	<i>512</i>

In tables we use a constant-width font for variable names and the names of structure members, and the slanted constant-width font for the names of defined constants. Here is an example:

<code>m_flags</code>	Description
<i>M_BCAST</i>	sent/received as link-level broadcast

We normally show all `#define` symbols this way. We show the value of the symbol if necessary (the value of `M_BCAST` is irrelevant) and sort the symbols alphabetically, unless some other ordering makes sense.

Throughout the text we'll use indented, parenthetical notes such as this to describe historical points or implementation minutiae.

We refer to Unix commands using the name of the command followed by a number in parentheses, as in `grep(1)`. The number in parentheses is the section number in the 4.4BSD manual of the "manual page" for the command, where additional information can be located.

## 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

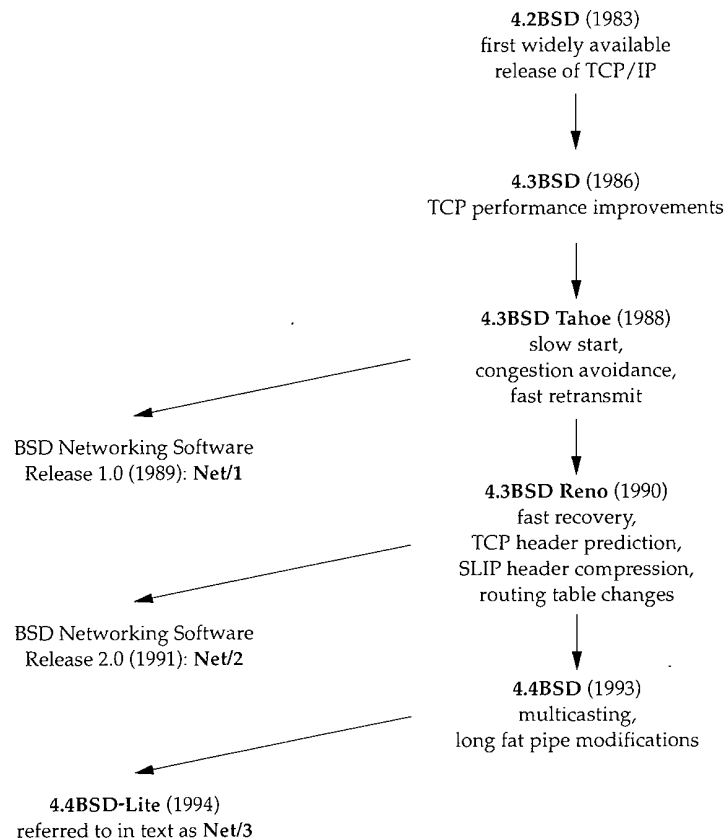


Figure 1.1 Various BSD releases with important TCP/IP features.

routines for the networking interface, and many of the applications and utilities (such as Telnet and FTP).

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.

## 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].

## 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.

```

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[BUFSIZE];
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, BUFSIZE, 0,
26             (struct sockaddr *) &serv, sizeof(serv)) != BUFSIZE)
27         err_sys("sendto error");

28     if ((n = recvfrom(sockfd, buff, BUFSIZE, 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 }

```

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

### 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 arbitrary—we 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`.

## 1.6 System Calls and Library Functions

All operating systems provide service points through which programs request services from the kernel. All variants of Unix provide a well-defined, limited number of kernel entry points known as *system calls*. We cannot change the system calls unless we have the kernel source code. Unix Version 7 provided about 50 system calls, 4.4BSD provides about 135, and SVR4 has around 120.

The system call interface is documented in Section 2 of the *Unix Programmer's Manual*. Its definition is in the C language, regardless of how system calls are invoked on any given system.

The Unix technique is for each system call to have a function of the same name in the standard C library. An application calls this function, using the standard C calling sequence. This function then invokes the appropriate kernel service, using whatever technique is required on the system. For example, the function may put one or more of the C arguments into general registers and then execute some machine instruction that generates a software interrupt into the kernel. For our purposes, we can consider the system calls to be C functions.

Section 3 of the *Unix Programmer's Manual* defines the general purpose functions available to programmers. These functions are not entry points into the kernel, although they may invoke one or more of the kernel's system calls. For example, the `printf` function may invoke the `write` system call to perform the output, but the functions `strcpy` (copy a string) and `atoi` (convert ASCII to integer) don't involve the operating system at all.

From an implementor's point of view, the distinction between a system call and a library function is fundamental. From a user's perspective, however, the difference is not as critical. For example, if we run Figure 1.2 under 4.4BSD, when the program calls the three functions `socket`, `sendto`, and `recvfrom`, each ends up calling a function of the same name within the kernel. We show the BSD kernel implementation of these three system calls later in the text.

If we run the program under SVR4, where the `socket` functions are in a user library that calls the "streams" subsystem, the interaction of these three functions with the kernel is completely different. Under SVR4 the call to `socket` ends up invoking the kernel's `open` system call for the file `/dev/udp` and then pushes the streams module `sockmod` onto the resulting stream. 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 in this text. We want to point out only that the implementation can be totally different while providing the same API to the application.

This difference in implementation technique also accounts for the manual page for the `socket` function appearing in Section 2 of the 4.4BSD manual but in Section 3n (the letter *n* stands for the networking subsection of Section 3) of the SVR4 manuals.

Finally, the implementation technique can change from one release to the next. For example, in Net/1 `send` and `sendto` were implemented as separate system calls within the kernel. In Net/3, however, `send` is a library function that calls `sendto`, which is a system call:

```
send(int s, char *msg, int len, int flags)
{
    return(sendto(s, msg, len, flags, (struct sockaddr *) NULL, 0));
}
```

The advantage in implementing `send` as a library function that just calls `sendto` is a reduction in the number of system calls and in the amount of code within the kernel. The disadvantage is the additional overhead of one more function call for the process that calls `send`.

Since this text describes the Berkeley implementation of TCP/IP, most of the functions called by the process (`socket`, `bind`, `connect`, etc.) are implemented directly in the kernel as system calls.

## 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.
2. 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).
3. 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.
4. 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.

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

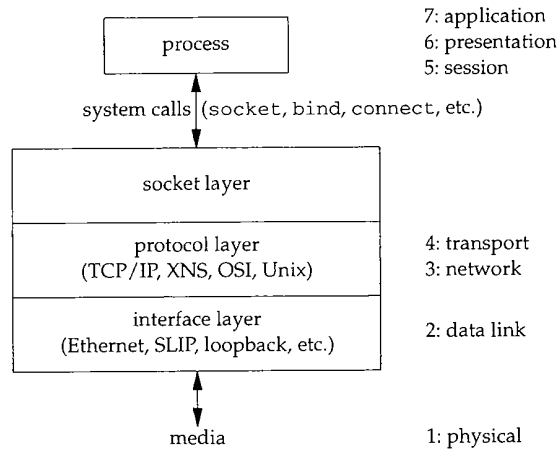


Figure 1.3 The general organization of networking code in Net/3.

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.

2. 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.
3. The *interface layer* contains the device drivers that communicate with the network devices.

## 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.

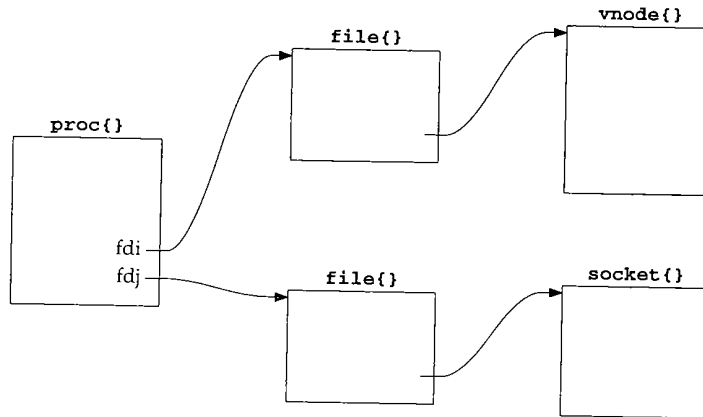


Figure 1.4 Fundamental relationship between kernel data structures starting with a descriptor.

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

```
a.out
```

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.

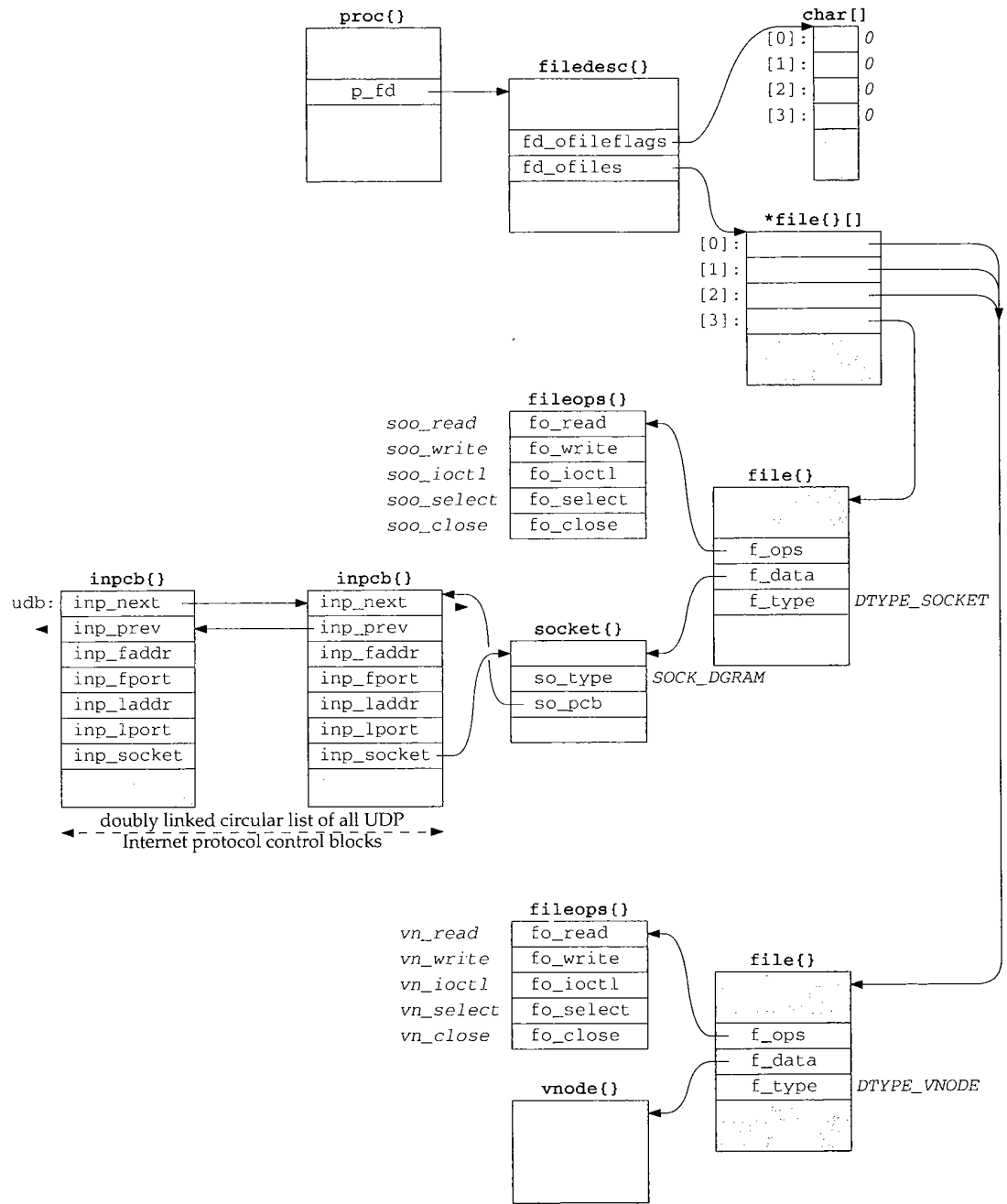


Figure 1.5 Kernel data structures after call to `socket` in example program.

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 `DTYPE_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`, `writew`, `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.
2. 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.
2. 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.
3. 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.



## 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.

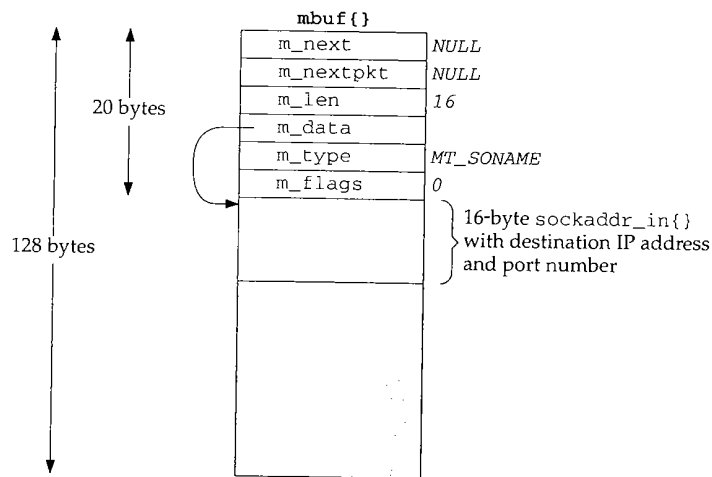


Figure 1.6 Mbuf containing destination address for `sendto`.

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_nextpkt` 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.

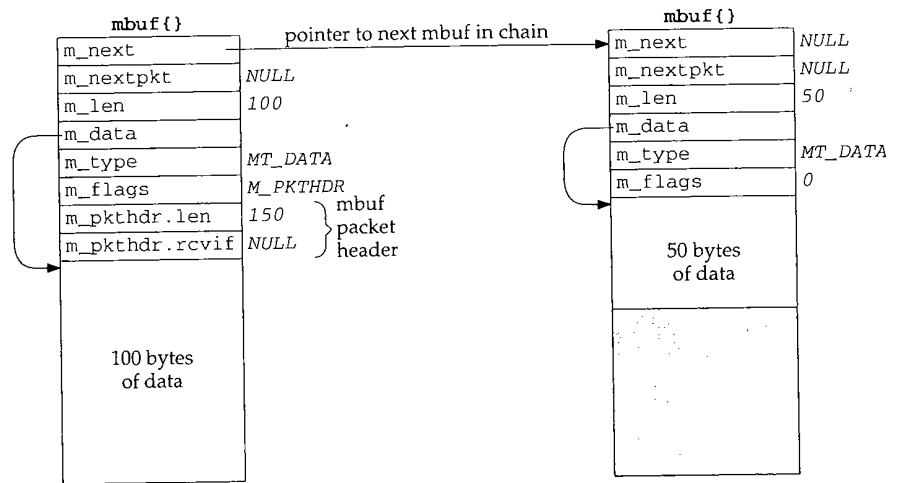


Figure 1.7 Two mbufs holding 150 bytes of data.

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.

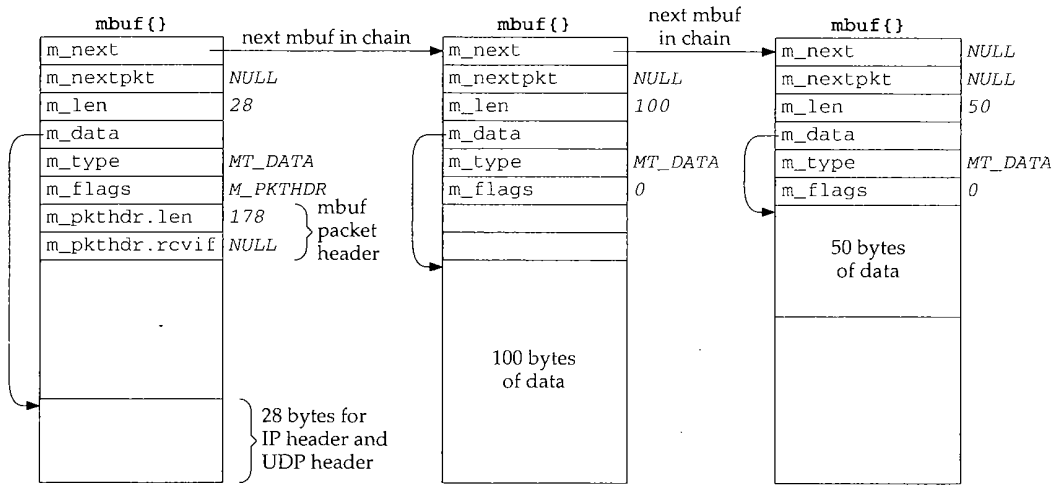


Figure 1.8 Mbuf chain from Figure 1.7 with another mbuf for IP and UDP headers prepended.

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.

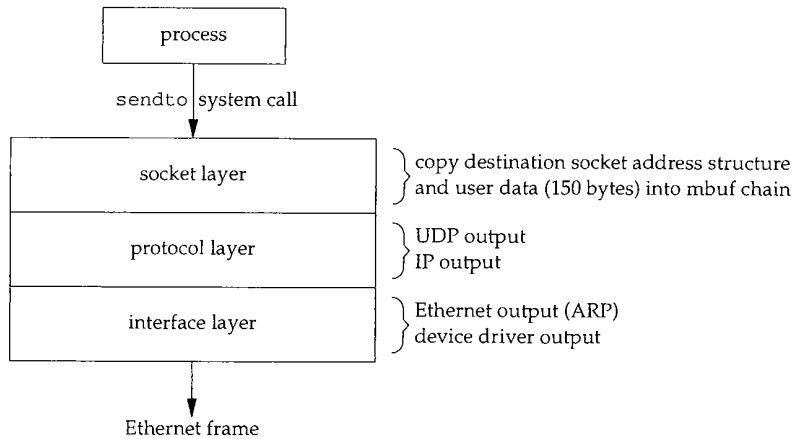


Figure 1.9 Processing performed by the three layers for simple UDP output.

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 queuing 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.

## 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.

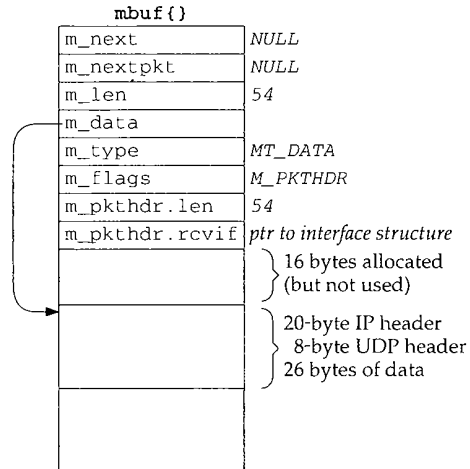


Figure 1.10 Single mbuf to hold input Ethernet data.

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 `rcvif` member contains a pointer to the interface structure corresponding to the received interface (Chapter 3). We see that the `rcvif` 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.

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`).

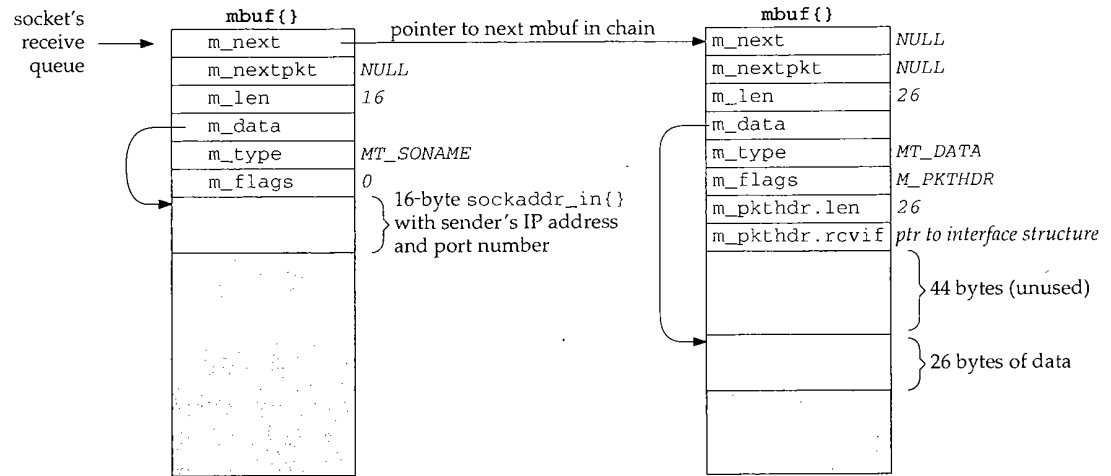


Figure 1.11 Sender's address and data.

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 run-able 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.

## 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.



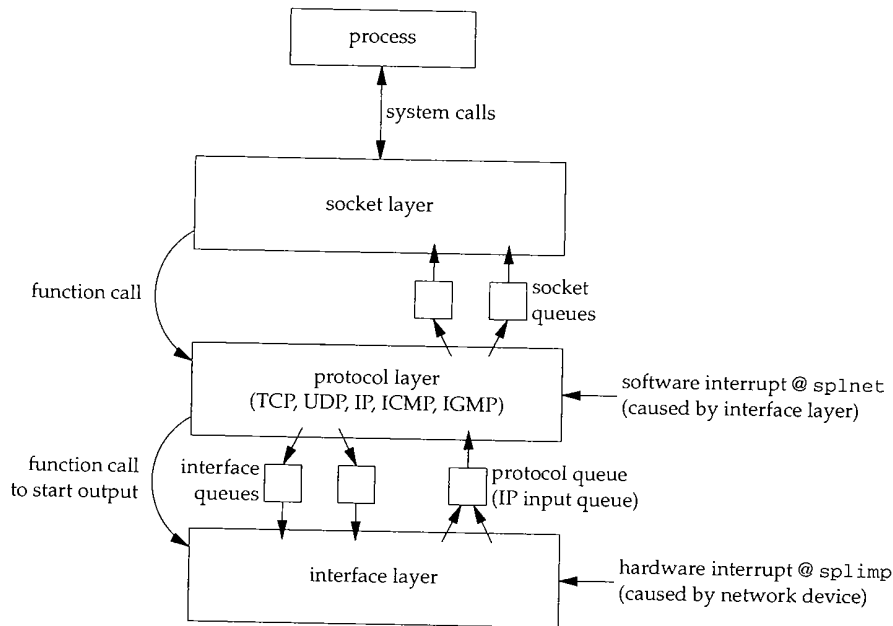


Figure 1.12 Communication between the layers for network input and output.

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.

## 1.12 Interrupt Levels and Concurrency

We saw in Section 1.10 that the processing of input packets by the networking code is asynchronous and interrupt driven. First, a device interrupt causes the interface layer code to execute, which posts a software interrupt that later causes the protocol layer code to execute. When the kernel is finished with these interrupt levels the socket code will execute.

There is a priority level assigned to each hardware and software interrupt. Figure 1.13 shows the normal ordering of the eight priority levels, from the lowest (no interrupts blocked) to the highest (all interrupts blocked).

Function	Description
sp10	normal operating mode, nothing blocked (lowest priority)
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 (highest priority)
splx(s)	(see text)

Figure 1.13 Kernel functions that block selected interrupts.

Table 4.5 of [Leffler et al. 1989] shows the priority levels used in the VAX implementation. The Net/3 implementation for the 386 uses the eight functions shown in Figure 1.13, but `splsoftclock` and `splnet` are at the same level, and `splclock` and `splhigh` are also at the same level.

The name *imp* that is used for the network interface level comes from the acronym IMP (Interface Message Processor), which was the original type of router used on the ARPANET.

The ordering of the different priority levels means that a higher-priority interrupt can preempt a lower-priority interrupt. Consider the sequence of events depicted in Figure 1.14.

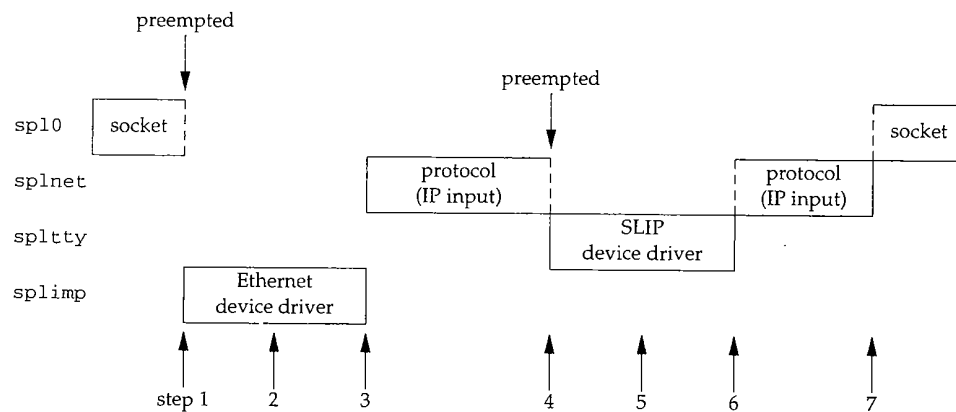


Figure 1.14 Example of priority levels and kernel processing.

1. While the socket layer is executing at `sp10`, an Ethernet device driver interrupt occurs, causing the interface layer to execute at `splimp`. This interrupt preempts the socket layer code. This is the asynchronous execution of the interface input routine.
2. While the Ethernet device driver is running, it places a received packet onto the IP input queue and schedules a software interrupt to occur at `splnet`. The

- software interrupt won't take effect immediately since the kernel is currently running at a higher priority level (*splimp*).
3. When the Ethernet device driver completes, the protocol layer executes at *splnet*. This is the asynchronous execution of the IP input routine.
  4. A terminal device interrupt occurs (say the completion of a SLIP packet) and it is handled immediately, preempting the protocol layer, since terminal I/O (*spltty*) is a higher priority than the protocol layer (*splnet*) in Figure 1.13. This is the asynchronous execution of the interface input routine.
  5. The SLIP driver places the received packet onto the IP input queue and schedules another software interrupt for the protocol layer.
  6. When the SLIP driver completes, the preempted protocol layer continues at *splnet*, finishes processing the packet received from the Ethernet device driver, and then processes the packet received from the SLIP driver. Only when there are no more input packets to process will it return control to whatever it preempted (the socket layer in this example).
  7. The socket layer continues from where it was preempted.

One concern with these different priority levels is how to handle data structures shared between the different levels. Examples of shared data structures are the three we show between the different levels in Figure 1.12—the socket, interface, and protocol queues. For example, while the IP input routine is taking a received packet off its input queue, a device interrupt can occur, preempting the protocol layer, and that device driver can add another packet to the IP input queue. These shared data structures (the IP input queue in this example, which is shared between the protocol layer and the interface layer) can be corrupted if nothing is done to coordinate the shared access.

The Net/3 code is sprinkled with calls to the functions *splimp* and *splnet*. These two calls are always paired with a call to *splx* to return the processor to the previous level. For example, here is the code executed by the IP input function at the protocol layer to check if there is another packet on its 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 level used by the network device drivers, preventing any network device driver interrupt from occurring. The previous priority level is returned as the value of the function and stored in the variable *s*. Then the macro *IF\_DEQUEUE* is executed to remove the next packet at the head of the IP input queue (*ipintrq*), placing the pointer to this mbuf chain in the variable *m*. Finally the CPU priority is returned to whatever it was when *splimp* was called, by calling *splx* with an argument of *s* (the saved value from the earlier call to *splimp*).

Since all network device driver interrupts are disabled between the calls to `splimp` and `splx`, the amount of code between these calls should be minimal. If interrupts are disabled for an extended period of time, additional device interrupts could be ignored, and data might be lost. For this reason the test of the variable `m` (to see if there is another packet to process) is performed after the call to `splx`, and not before the call.

The Ethernet output routine needs these `spl` calls when it places an outgoing packet onto an interface's queue, tests whether the interface is currently busy, and starts the interface if it was not busy.

```

struct mbuf *m;
int s;

s = splimp();
/*
 * Queue message on interface, and start output if interface not active.
 */
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd); /* queue is full, drop packet */
    splx(s);
    error = ENOBUFS;
    goto bad;
}

IF_ENQUEUE(&ifp->if_snd, m); /* add the packet to interface queue */

if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp); /* start interface */

splx(s);

```

The reason device interrupts are disabled in this example is to prevent the device driver from taking the next packet off its send queue while the protocol layer is adding a packet to that queue. The driver's send queue is a data structure shared between the protocol layer and the interface layer.

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

### 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.

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.

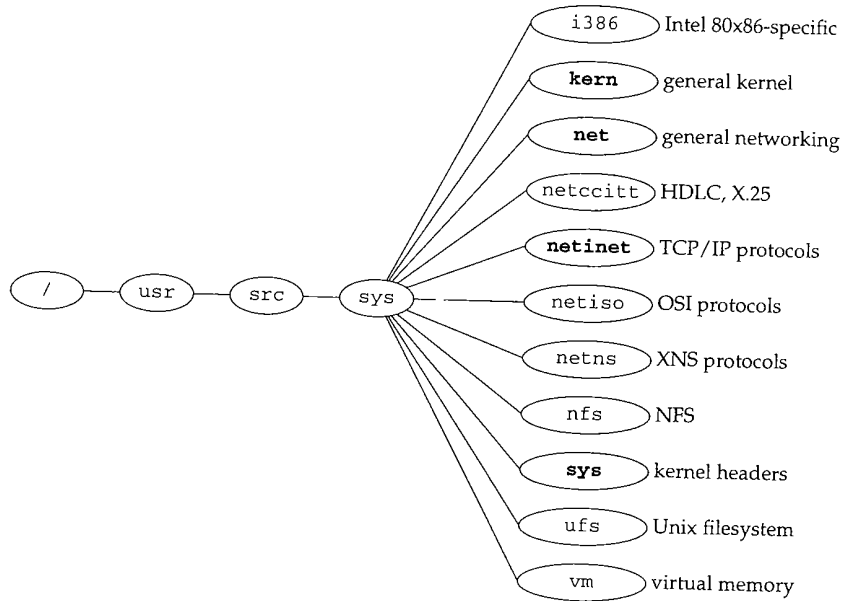


Figure 1.15 Net/3 source code organization.

- **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 directory—the 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.

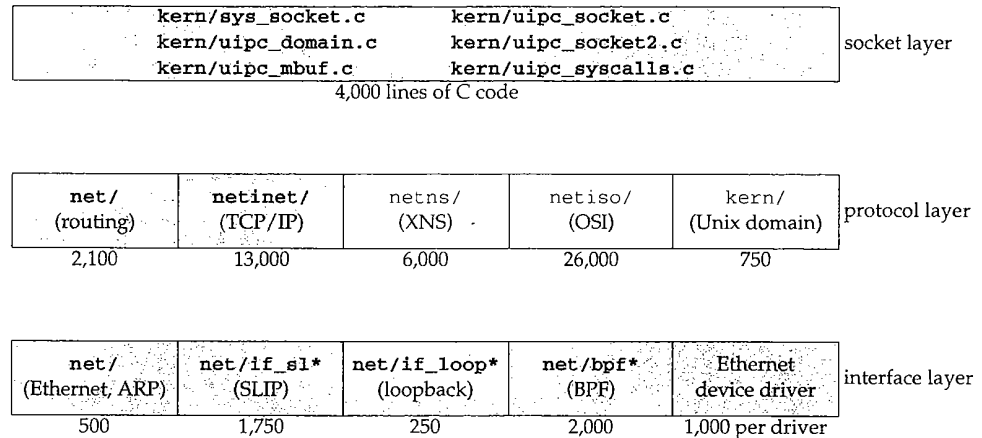


Figure 1.16 Net/3 source code organization mapped to three kernel layers.

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.

## 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.

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.

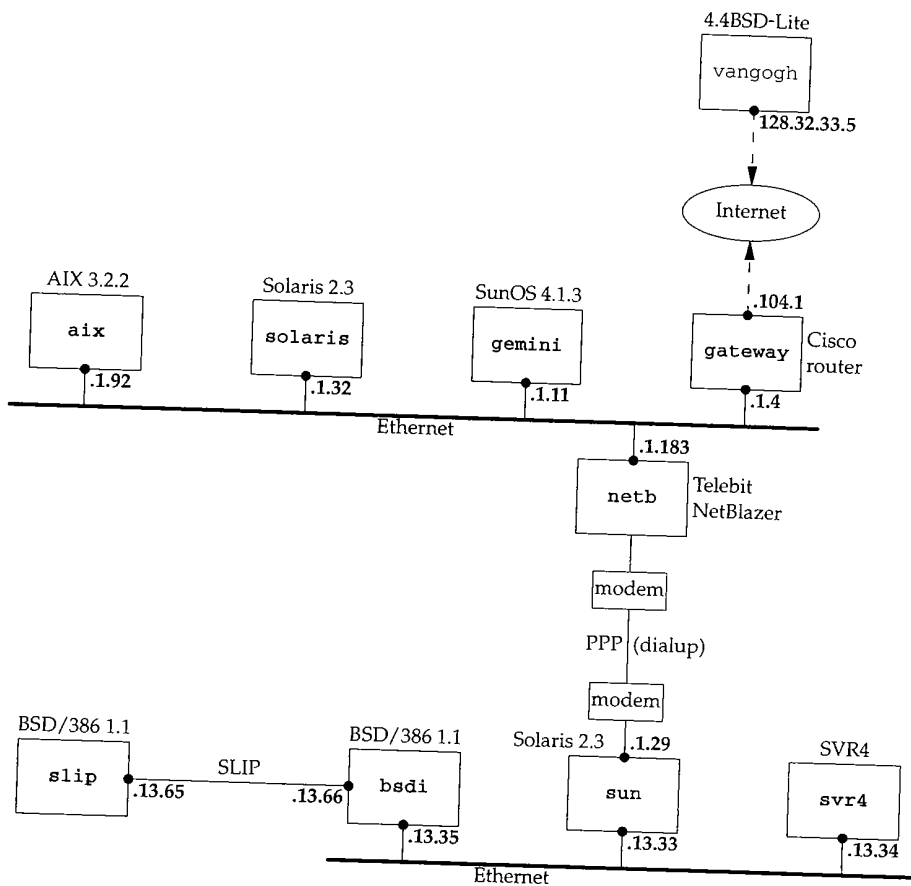


Figure 1.17 Test network used for all the examples in the text.

### 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 `sp1` 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

- 1.1 Type in the example program (Figure 1.2) and run it on your system. If your system has a system call tracing capability, such as `trace` (SunOS 4.x), `truss` (SVR4), or `ktrace` (4.4BSD), use it to determine the system calls invoked by this example.
- 1.2 In our example that calls `IF_DEQUEUE` in Section 1.12, we noted that the call to `splimp` blocks network device drivers from interrupting. While Ethernet drivers execute at this level, what happens to SLIP drivers?



# 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:

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_data` 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.

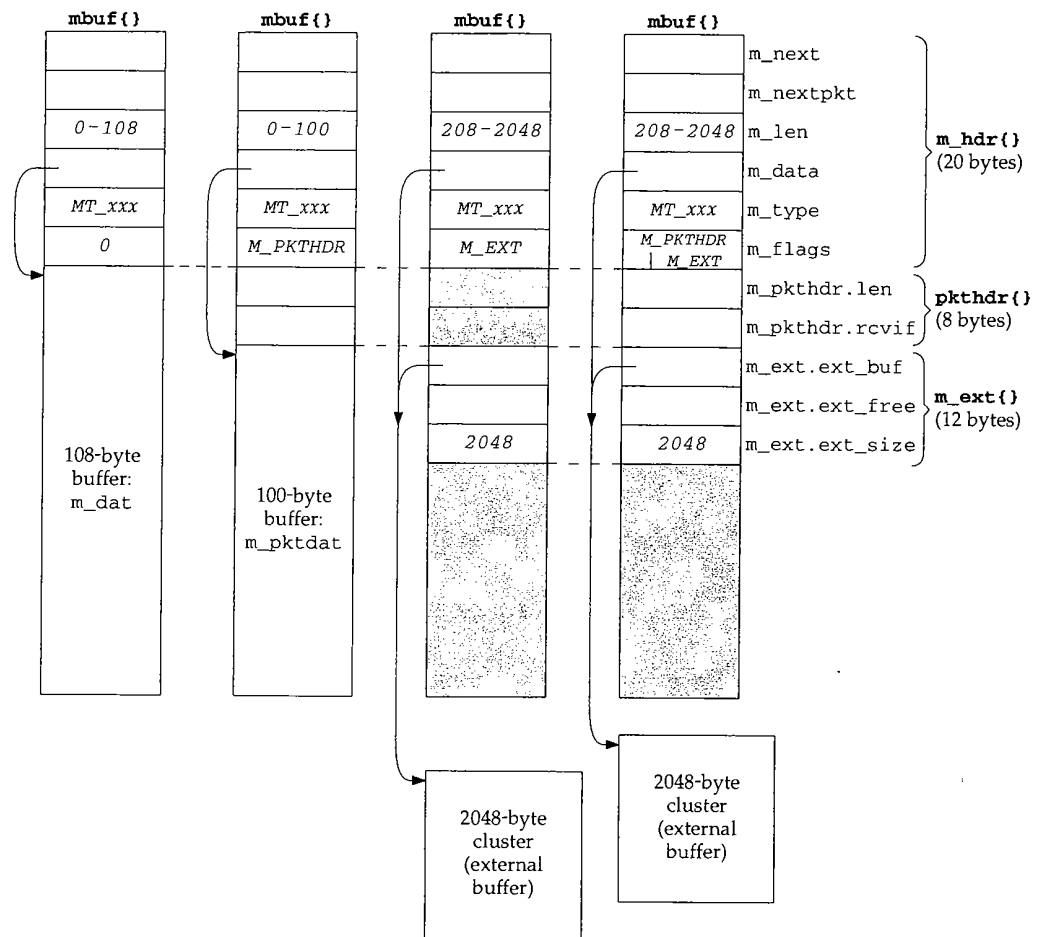


Figure 2.1 Four different types of mbufs, depending on the `m_flags` value.

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.

2. 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_pktdata` 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).

3. 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.

4. 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.

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).

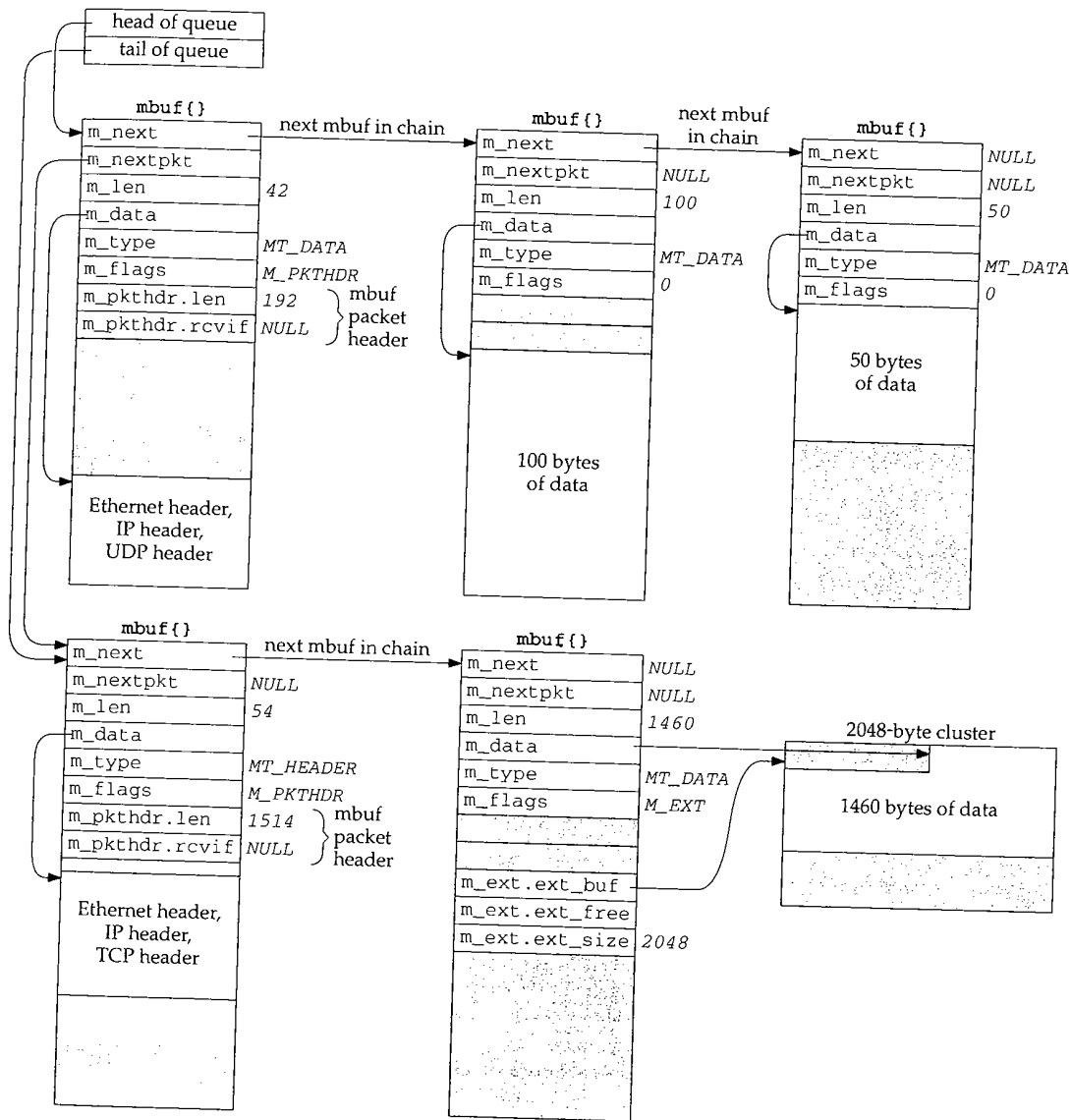


Figure 2.2 Two packets on a queue: first with 192 bytes of data and second with 1514 bytes of data.

## 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.

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

Figure 2.3 Files discussed in this chapter.

### Global Variables

One global variable is introduced in this chapter, shown in Figure 2.4.

Variable	Datatype	Description
mbstat	struct mbstat	mbuf statistics (Figure 2.5)

Figure 2.4 Global variables introduced in this chapter.

### Statistics

Various statistics are maintained in the global structure `mbstat`, described in Figure 2.5.

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

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

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).

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.

netstat -m output	mbstat member
99 mbufs in use:	
1 mbufs allocated to data	m_mtypes[MT_DATA]
43 mbufs allocated to packet headers	m_mtypes[MT_HEADER]
17 mbufs allocated to protocol control blocks	m_mtypes[MT_PCB]
20 mbufs allocated to socket names and addresses	m_mtypes[MT_SONAME]
18 mbufs allocated to socket options	m_mtypes[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

Figure 2.6 Sample mbuf statistics.

### 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.

### 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.

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

Figure 2.7 Mbuf constants from mbuf.h.

## 2.4 mbuf Structure

Figure 2.8 shows the definition of the mbuf structure.

```

60 /* header at beginning of each mbuf: */
61 struct m_hdr {
62     struct mbuf *mh_next;      /* next buffer in chain */
63     struct mbuf *mh_nextpkt;  /* next chain in queue/record */
64     int mh_len;               /* amount of data in this mbuf */
65     caddr_t mh_data;         /* pointer to data */
66     short mh_type;          /* type of data (Figure 2.10) */
67     short mh_flags;         /* flags (Figure 2.9) */
68 };

69 /* record/packet header in first mbuf of chain; valid if M_PKTHDR set */
70 struct pkthdr {
71     int len;                 /* total packet length */
72     struct ifnet *rcvif;    /* receive interface */
73 };

74 /* description of external storage mapped into mbuf, valid if M_EXT set */
75 struct m_ext {
76     caddr_t ext_buf;        /* start of buffer */
77     void (*ext_free) ();    /* free routine if not the usual */
78     u_int ext_size;        /* size of buffer, for ext_free */
79 };

80 struct mbuf {
81     struct m_hdr m_hdr;
82     union {
83         struct {
84             struct pkthdr MH_pkthdr; /* M_PKTHDR set */
85             union {
86                 struct m_ext MH_ext; /* M_EXT set */
87                 char MH_databuf[MHLEN];
88             } MH_dat;
89         } MH;
90         char M_databuf[MLEN]; /* !M_PKTHDR, !M_EXT */
91     } M_dat;
92 };

93 #define m_next      m_hdr.mh_next
94 #define m_len      m_hdr.mh_len
95 #define m_data     m_hdr.mh_data
96 #define m_type     m_hdr.mh_type
97 #define m_flags    m_hdr.mh_flags
98 #define m_nextpkt  m_hdr.mh_nextpkt
99 #define m_act      m_nextpkt
100 #define m_pkthdr   M_dat.MH.MH_pkthdr
101 #define m_ext      M_dat.MH.MH_dat.MH_ext
102 #define m_pktdat   M_dat.MH.MH_dat.MH_databuf
103 #define m_dat      M_dat.M_databuf

```

Figure 2.8 Mbuf structures.



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.

m_flags	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>

Figure 2.9 m\_flags values.

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`.

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

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

Figure 2.10 Values for m\_type member.

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.

## 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`.

```

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 }

```

— uipc\_mbuf.c

— uipc\_mbuf.c

Figure 2.11 `m_get` function: allocate an mbuf.

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

```

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'll see in Figure 2.13, waiting for an mbuf does not guarantee that one will be available.

```

154 #define MGET(m, how, type) { \
155     MALLOC(m, struct mbuf *, MSIZE, mtypes[type], (how)); \
156     if (m) { \
157         (m)->m_type = (type); \
158         MBUFLOCK(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 }

```

*mbuf.h*

*mbuf.h*

Figure 2.12 MGET macro.

154-157 MGET first calls the kernel's `MALLOC` macro, which is the general-purpose kernel memory allocator. The array `mtypes` 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_mtypes[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 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

uipc\_mbuf.c

Figure 2.13 `m_retry` function.

<sup>92-97</sup> 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.

<sup>98-102</sup> 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 `m` 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 at in this section, other than the call to `MBUFLOCK` 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 macros 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`.

## 2.6 m\_devget and m\_pullup Functions

We encounter the `m_pullup` function when we show the code for IP, ICMP, IGMP, UDP, and TCP. It is called to guarantee that the specified number of bytes (the size of the corresponding protocol header) are contiguous in the first mbuf of a chain; otherwise the specified number of bytes are copied to a new mbuf and made contiguous. To understand the usage of `m_pullup` we must describe its implementation and its interaction with both the `m_devget` function and the `mtod` and `dtom` macros. This description also provides additional insight into the usage of mbufs in Net/3.

### m\_devget Function

When an Ethernet frame is received, the device driver calls the function `m_devget` to create an mbuf chain and copy the frame from the device into the chain. Depending on the length of the received frame (excluding the Ethernet header), there are four different possibilities for the resulting mbuf chain. The first two possibilities are shown in Figure 2.14.

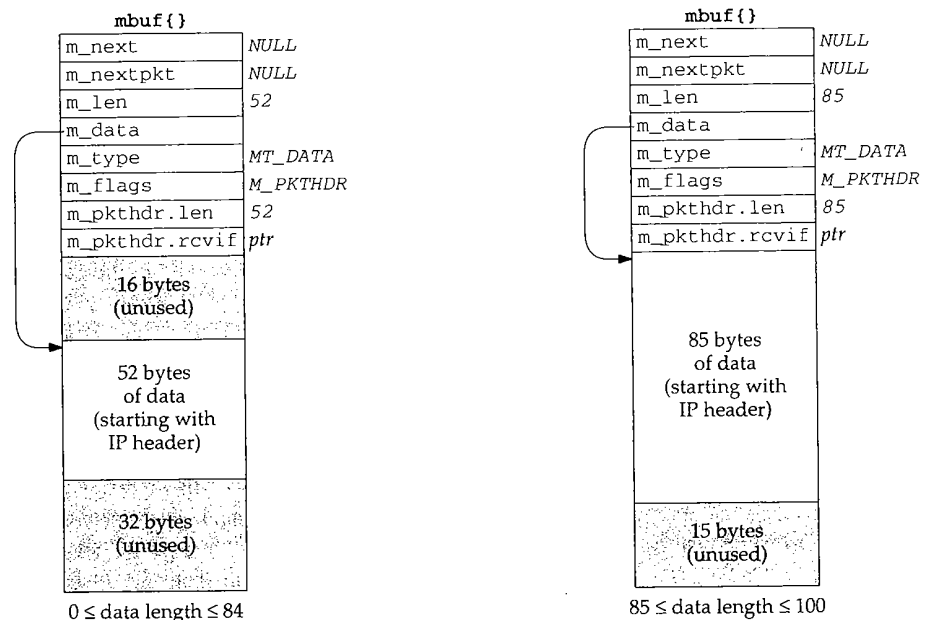


Figure 2.14 First two types of mbufs created by `m_devget`.

1. The left mbuf in Figure 2.14 is used when the amount of data is between 0 and 84 bytes. In this figure we assume there are 52 bytes of data: a 20-byte IP header and a 32-byte TCP header (the standard 20-byte TCP header plus 12 bytes of TCP options)

but no TCP data. Since the data in the mbuf returned by `m_devget` starts with the IP header, the realistic minimum value for `m_len` is 28: 20 bytes for an IP header, 8 bytes for a UDP header, and a 0-length UDP datagram.

`m_devget` leaves 16 bytes unused at the beginning of the mbuf. Although the 14-byte Ethernet header is not stored here, room is allocated for a 14-byte Ethernet header on output, should the same mbuf be used for output. We'll encounter two functions that generate a response by using the received mbuf as the outgoing mbuf: `icmp_reflect` and `tcp_respond`. In both cases the size of the received datagram is normally less than 84 bytes, so it costs nothing to leave room for 16 bytes at the front, which saves time when building the outgoing datagram. The reason 16 bytes are allocated, and not 14, is to have the IP header longword aligned in the mbuf.

2. If the amount of data is between 85 and 100 bytes, the data still fits in a packet header mbuf, but there is no room for the 16 bytes at the beginning. The data starts at the beginning of the `m_pktdat` array and any unused space is at the end of this array. The mbuf on the right in Figure 2.14 shows this example, assuming 85 bytes of data.
3. Figure 2.15 shows the third type of mbuf created by `m_devget`. Two mbufs are required when the amount of data is between 101 and 207 bytes. The first 100 bytes are stored in the first mbuf (the one with the packet header), and the remainder are stored in the second mbuf. In this example we show a 104-byte datagram. No attempt is made to leave 16 bytes at the beginning of the first mbuf.

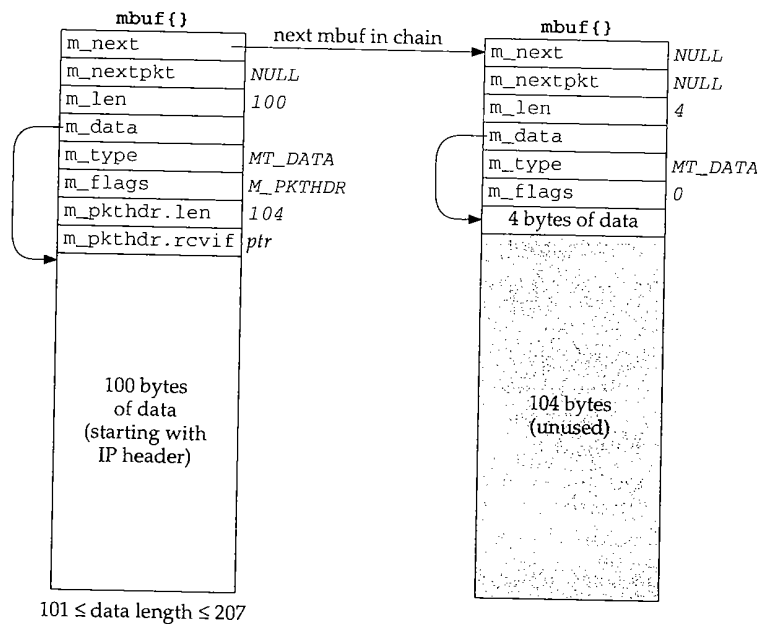


Figure 2.15 Third type of mbuf created by `m_devget`.

4. Figure 2.16 shows the fourth type of mbuf created by `m_devget`. If the amount of data is greater than or equal to 208 (`MINCLBYTES`), one or more clusters are used. The example in the figure assumes a 1500-byte Ethernet frame with 2048-byte clusters. If 1024-byte clusters are in use, this example would require two mbufs, each with the `M_EXT` flag set, and each pointing to a cluster.

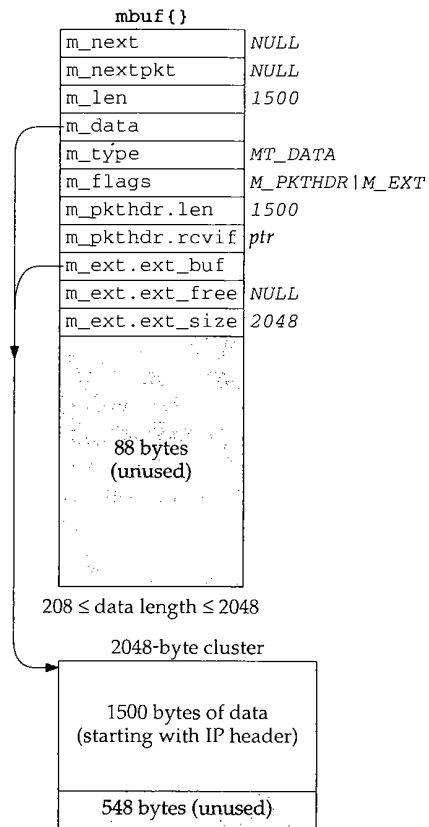


Figure 2.16 Fourth type of mbuf created by `m_devget`.

#### mtod and dtom Macros

The two macros `mtod` and `dtom` are also defined in `mbuf.h`. They simplify complex mbuf structure expressions.

```
#define mtod(m,t) ((t)((m)->m_data))
#define dtom(x) ((struct mbuf*)((int)(x) & ~(MSIZE-1)))
```

`mtod` ("mbuf-to-data") returns a pointer to the data associated with an mbuf, and casts the pointer to a specified type. For example, the code



```

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 type cast is required by the C compiler and the code then references the IP header using the pointer `ip`. We see this macro used when a C structure (often a protocol header) is stored in an mbuf. This macro works if the data is stored in the mbuf itself (Figures 2.14 and 2.15) or if the data is stored in a cluster (Figure 2.16).

The macro `dtom` ("data-to-mbuf") takes a pointer to data anywhere within the data portion of the mbuf and returns a pointer to the mbuf structure itself. For example, if we know that `ip` points within the data area of an mbuf, the sequence

```

struct mbuf *m;
struct ip *ip;

m = dtom(ip);

```

stores the pointer to the beginning of the mbuf in `m`. By knowing that `MSIZE` (128) is a power of 2, and that mbufs are always aligned by the kernel's memory allocator on `MSIZE` byte blocks of memory, `dtom` just clears the appropriate low-order bits in its argument pointer to find the beginning of the mbuf.

There is a problem with `dtom`: it doesn't work if its argument points to a cluster, or within a cluster, as in Figure 2.16. Since there is no pointer from the cluster back to the mbuf structure, `dtom` cannot be used. This leads to the next function, `m_pullup`.

### **m\_pullup Function and Contiguous Protocol Headers**

The `m_pullup` function has two purposes. The first is when one of the protocols (IP, ICMP, IGMP, UDP, or TCP) finds that the amount of data in the first mbuf (`m_len`) is less than the size of the minimum protocol header (e.g., 20 for IP, 8 for UDP, 20 for TCP). `m_pullup` is called on the assumption that the remaining part of the header is in the next mbuf on the chain. `m_pullup` rearranges the mbuf chain so that the first `N` bytes of data are contiguous in the first mbuf on the chain. `N` is an argument to the function that must be less than or equal to 100 (MHLLEN). If the first `N` bytes are contiguous in the first mbuf, then both of the macros `mtod` and `dtom` will work.

For example, we'll encounter the following code in the IP input routine:

```

if (m->m_len < sizeof(struct ip) &&
    (m = m_pullup(m, sizeof(struct ip))) == 0) {
    ipstat.ips_toosmall++;
    goto next;
}
ip = mtod(m, struct ip *);

```

If the amount of data in the first mbuf is less than 20 (the size of the standard IP header), `m_pullup` is called. `m_pullup` can fail for two reasons: (1) if it needs another mbuf

and its call to `MGET` fails, or (2) if the total amount of data in the mbuf chain is less than the requested number of contiguous bytes (what we called  $N$ , which in this case is 20). The second reason is the most common cause of failure. In this example, if `m_pullup` fails, an IP counter is incremented and the IP datagram is discarded. Notice that this code assumes the reason for failure is that the amount of data in the mbuf chain is less than 20 bytes.

In actuality, `m_pullup` is rarely called in this scenario (notice that `C's` `&&` operator only calls it when the mbuf length is smaller than expected) and when it is called, it normally fails. The reason can be seen by looking at Figure 2.14 through Figure 2.16: there is room in the first mbuf, or in the cluster, for at least 100 contiguous bytes, starting with the IP header. This allows for the maximum IP header of 60 bytes followed by 40 bytes of TCP header. (The other protocols—ICMP, IGMP, and UDP—have headers smaller than 40 bytes.) If the data bytes are available in the mbuf chain (the packet is not smaller than the minimum required by the protocol), then the required number of bytes should always be contiguous in the first mbuf. But if the received packet is too short (`m_len` is less than the expected minimum), then `m_pullup` is called and it returns an error, since the required amount of data is not available in the mbuf chain.

Berkeley-derived kernels maintain a variable named `MPPFail` that is incremented each time `m_pullup` fails. On a Net/3 system that had received over 27 million IP datagrams, `MPPFail` was 9. The counter `ipstat.ips_toosmall` was also 9 and all the other protocol counters (i.e., ICMP, IGMP, UDP, and TCP) following a failure of `m_pullup` were 0. This confirms our statement that most failures of `m_pullup` are because the received IP datagram was too small.

### `m_pullup` and IP Fragmentation and Reassembly

The second use of `m_pullup` concerns IP reassembly and TCP reassembly. Assume IP receives a packet of length 296, which is a fragment of a larger IP datagram. The mbuf passed from the device driver to IP input looks like the one we showed in Figure 2.16: the 296 bytes of data are stored in a cluster. We show this in Figure 2.17.

The problem is that the IP fragmentation algorithm keeps the individual fragments on a doubly linked list, using the source and destination IP address fields in the IP header to hold the forward and backward list pointers. (These two IP addresses are saved, of course, in the head of the list, since they must be put back into the reassembled datagram. We describe this in Chapter 10.) But if the IP header is in a cluster, as shown in Figure 2.17, these linked list pointers would be in the cluster, and when the list is traversed at some later time, the pointer to the IP header (i.e., the pointer to the beginning of the cluster) could not be converted into the pointer to the mbuf. This is the problem we mentioned earlier in this section: the `dtom` macro cannot be used if `m_data` points into a cluster, because there is no back pointer from the cluster to the mbuf. IP fragmentation cannot store the links in the cluster as shown in Figure 2.17.

To solve this problem the IP fragmentation routine *always* calls `m_pullup` when a fragment is received, if the fragment is contained in a cluster. This forces the 20-byte IP header into its own mbuf. The code looks like

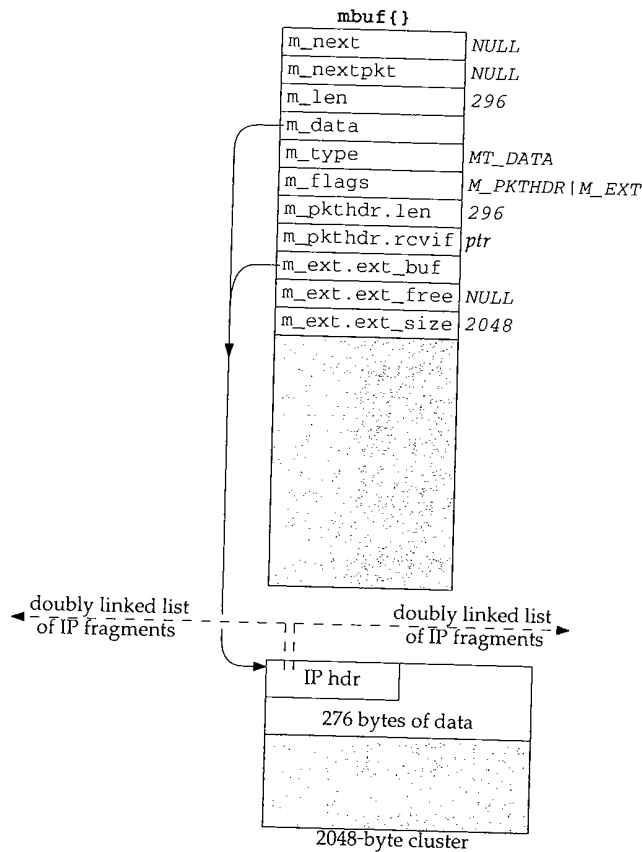


Figure 2.17 An IP fragment of length 296.

```

if (m->m_flags & M_EXT) {
    if ((m = m_pullup(m, sizeof(struct ip))) == 0) {
        ipstat.ips_toosmall++;
        goto next;
    }
    ip = mtd(m, struct ip *);
}

```

Figure 2.18 shows the resulting mbuf chain, after m\_pullup is called. m\_pullup allocates a new mbuf, prepends it to the chain, and moves the first 40 bytes of data from the cluster into the new mbuf. The reason it moves 40 bytes, and not just the requested 20, is to try to save an additional call at a later time when IP passes the datagram to a higher-layer protocol (e.g., ICMP, IGMP, UDP, or TCP). The magic number 40 (max\_protohdr in Figure 7.17) is because the largest protocol header normally encountered is the combination of a 20-byte IP header and a 20-byte TCP header. (This

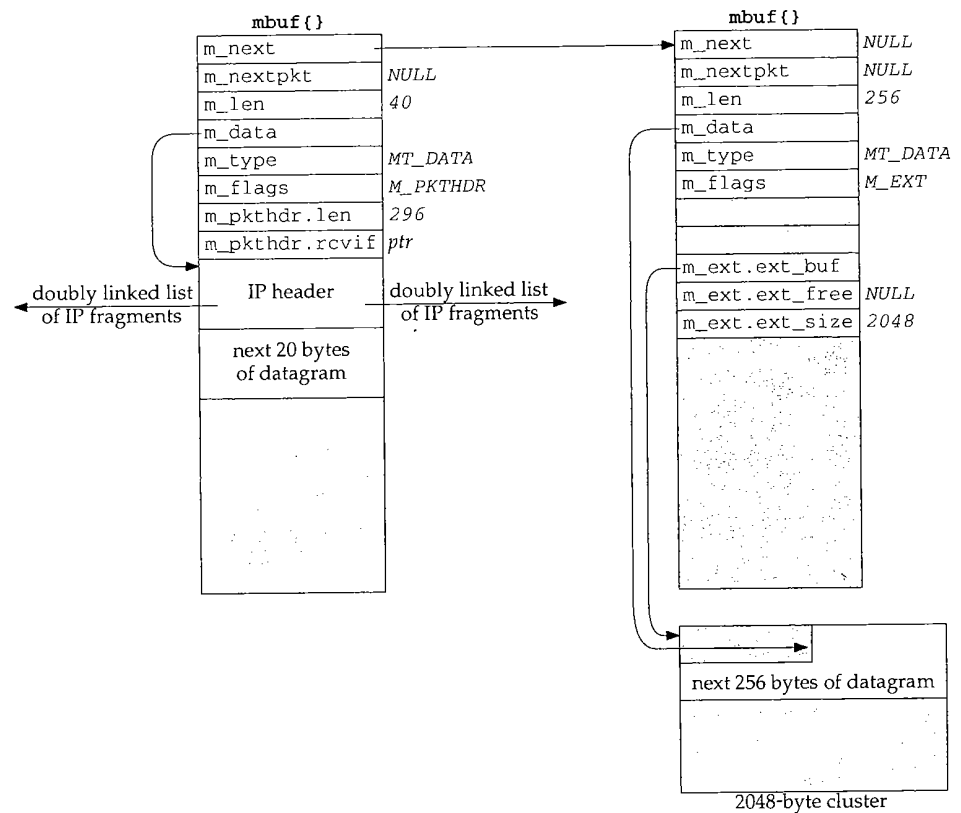


Figure 2.18 An IP fragment of length 296, after calling `m_pullup`.

assumes that other protocol suites, such as the OSI protocols, are not compiled into the kernel.)

In Figure 2.18 the IP fragmentation algorithm can save a pointer to the IP header contained in the mbuf on the left, and this pointer can be converted into a pointer 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 technique to avoid calling `m_pullup`. This is because `m_pullup` is expensive: memory is allocated and data is copied from a cluster to an mbuf. TCP tries to avoid data copying whenever possible.

Chapter 19 of Volume 1 mentions that about one-half of TCP data is bulk data (often 512 or more bytes of data per segment) and the other half is interactive data (of which about 90% of the segments contain less than 10 bytes of data). Hence, when TCP receives segments from IP they are usually in the format shown on the left of Figure 2.14 (a small amount of interactive data, stored in the mbuf itself) or in the format shown in

Figure 2.16 (bulk data, stored in a cluster). When TCP segments arrive out of order, they are stored on a doubly linked list by TCP. As with IP fragmentation, fields in the IP header are used to hold the list pointers, which is OK since these fields are no longer needed once the IP datagram is accepted by TCP. But the same problem arises with the conversion of a list pointer into the corresponding mbuf pointer, when the IP header is stored in a cluster (Figure 2.17).

To solve the problem, we'll see in Section 27.9 that TCP stores the mbuf pointer in some unused fields in the TCP header, providing a back pointer of its own from the cluster to the mbuf, just to avoid calling `m_pullup` for every out-of-order segment. If the IP header is contained in the data portion of the mbuf (Figure 2.18), then this back pointer is superfluous, since the `dtom` macro would work on the list pointer. But if the IP header is contained in a cluster, this back pointer is required. We'll examine the source code that implements this technique when we describe `tcp_reass` in Section 27.9.

### Summary of `m_pullup` Usage

We've described three main points about `m_pullup`.

- Most device drivers do not split the first portion of an IP datagram between mbufs. Therefore the possible calls to `m_pullup` that we'll encounter in every protocol (IP, ICMP, IGMP, UDP, and TCP), just to assure that the protocol header is stored contiguously, rarely take place. When these calls to `m_pullup` do occur, it is normally because the IP datagram is too small, in which case `m_pullup` returns an error, the datagram is discarded, and an error counter is incremented.
- `m_pullup` is called for every received IP fragment, when the IP fragment is stored in a cluster. This means that `m_pullup` is called for almost every received fragment, since the length of most fragments is greater than 208 bytes.
- As long as TCP segments are not fragmented by IP, the receipt of a TCP segment, whether it be in order or out of order, should not invoke `m_pullup`. This is one reason to avoid IP fragmentation with TCP.

## 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.

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.

Macro	Description
<code>MCLGET</code>	Get a cluster (an external buffer) and set the data pointer ( <code>m_data</code> ) of the existing mbuf pointed to by <i>m</i> 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>
<code>MFREE</code>	Free the single mbuf pointed to by <i>m</i> . If <i>m</i> 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 <i>m</i> 's successor (pointed to by <code>m-&gt;m_next</code> , which can be null) is stored in <i>n</i> .  <code>void MFREE(struct mbuf *m, struct mbuf *n);</code>
<code>MGETHDR</code>	Allocate an mbuf and initialize it as a packet header. This macro is similar to <code>MGET</code> (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>
<code>MH_ALIGN</code>	Set the <code>m_data</code> pointer of an mbuf containing a packet header to provide room for an object of size <i>len</i> 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>
<code>M_PREPEND</code>	Prepend <i>len</i> bytes of data in front of the data in the mbuf pointed to by <i>m</i> . If room exists in the mbuf, just decrement the pointer ( <code>m_data</code> ) and increment the length ( <code>m_len</code> ) by <i>len</i> bytes. If there is not enough room, a new mbuf is allocated, its <code>m_next</code> pointer is set to <i>m</i> , a pointer to the new mbuf is stored in <i>m</i> , and the data pointer of the new mbuf is set so that the <i>len</i> 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_PREPEND(struct mbuf *m, int len, int nowait);</code>
<code>dtom</code>	Convert the pointer <i>x</i> , 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>
<code>mtod</code>	Type cast the pointer to the data area of the mbuf pointed to by <i>m</i> to <i>type</i> .  <code>type mtod(struct mbuf *m, type);</code>

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

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.

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. void <b>m_adj</b> (struct mbuf * <i>m</i> , int <i>len</i> );
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). void <b>m_cat</b> (struct mbuf * <i>m</i> , struct mbuf * <i>n</i> );
m_copy	A three-argument version of <b>m_copy</b> that implies a fourth argument of <b>M_DONTWAIT</b> . struct mbuf * <b>m_copy</b> (struct mbuf * <i>m</i> , int <i>offset</i> , int <i>len</i> );
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. void <b>m_copydata</b> (struct mbuf * <i>m</i> , int <i>offset</i> , int <i>len</i> , caddr_t <i>cp</i> );
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. void <b>m_copyback</b> (struct mbuf * <i>m</i> , int <i>offset</i> , int <i>len</i> , caddr_t <i>cp</i> );
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 <b>M_COPYALL</b> , the remainder of the mbuf chain starting at <i>offset</i> is copied. We say more about this function in Section 2.9. struct mbuf * <b>m_copym</b> (struct mbuf * <i>m</i> , int <i>offset</i> , int <i>len</i> , int <i>nowait</i> );
m_devget	Create a new mbuf chain with a packet header and return the pointer to the chain. The <i>len</i> and <i>rcvif</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. struct mbuf * <b>m_devget</b> (char * <i>buf</i> , int <i>len</i> , int <i>off</i> , struct ifnet * <i>ifp</i> , void (* <i>copy</i> )(const void *, void *, u_int));
m_free	A function version of the macro <b>MFREE</b> . struct mbuf * <b>m_free</b> (struct mbuf * <i>m</i> );
m_freem	Free all the mbufs in the chain pointed to by <i>m</i> . void <b>m_freem</b> (struct mbuf * <i>m</i> );
m_get	A function version of the <b>MGET</b> macro. We showed this function in Figure 2.12. struct mbuf * <b>m_get</b> (int <i>nowait</i> , int <i>type</i> );
m_getclr	This function calls the <b>MGET</b> macro to get an mbuf and then zeros the 108-byte buffer. struct mbuf * <b>m_getclr</b> (int <i>nowait</i> , int <i>type</i> );
m_gethdr	A function version of the <b>MGETHDR</b> macro. struct mbuf * <b>m_gethdr</b> (int <i>nowait</i> , int <i>type</i> );
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 <b>mtod</b> macro returns a pointer that correctly references a structure of size <i>len</i> . We described this function in Section 2.6. struct mbuf * <b>m_pullup</b> (struct mbuf * <i>m</i> , int <i>len</i> );

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

## 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.
2. 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.

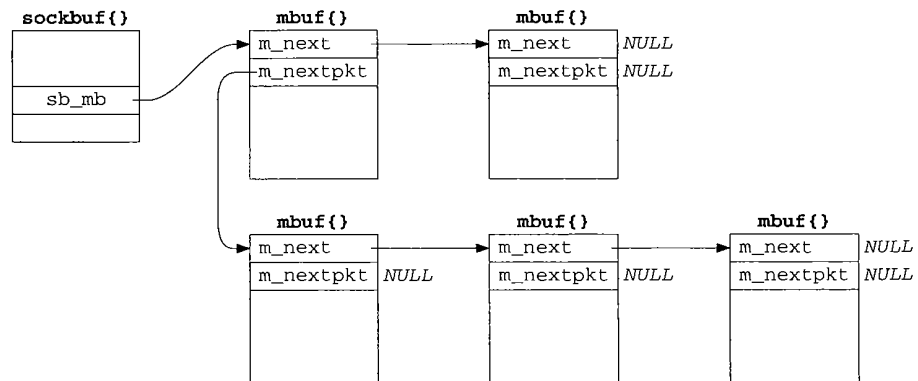


Figure 2.21 Linked list of mbuf chains with head pointer only.

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.

3. 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.

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



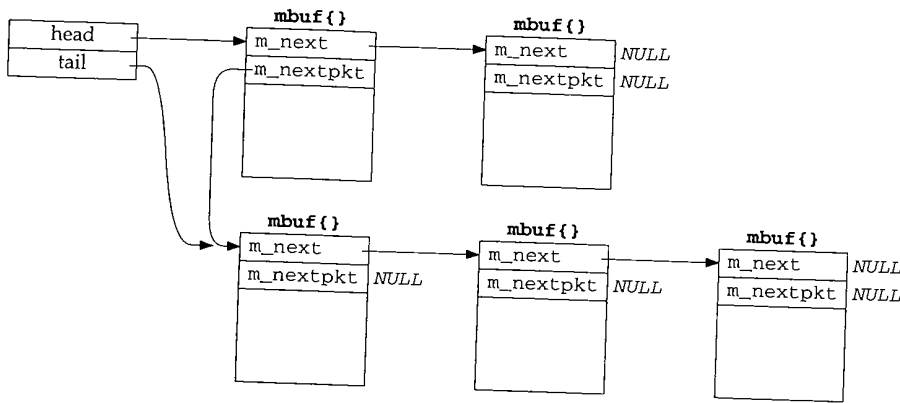


Figure 2.22 Linked list with head and tail pointers.

4. 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).

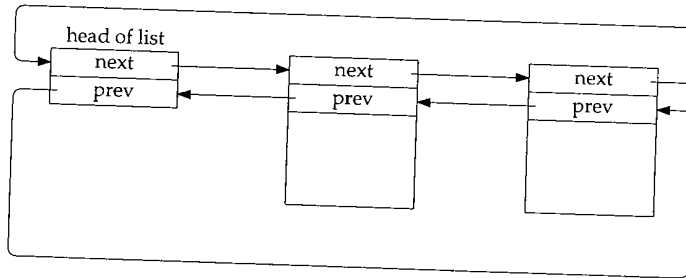


Figure 2.23 Doubly linked, circular list.

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.

## 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 × 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`.

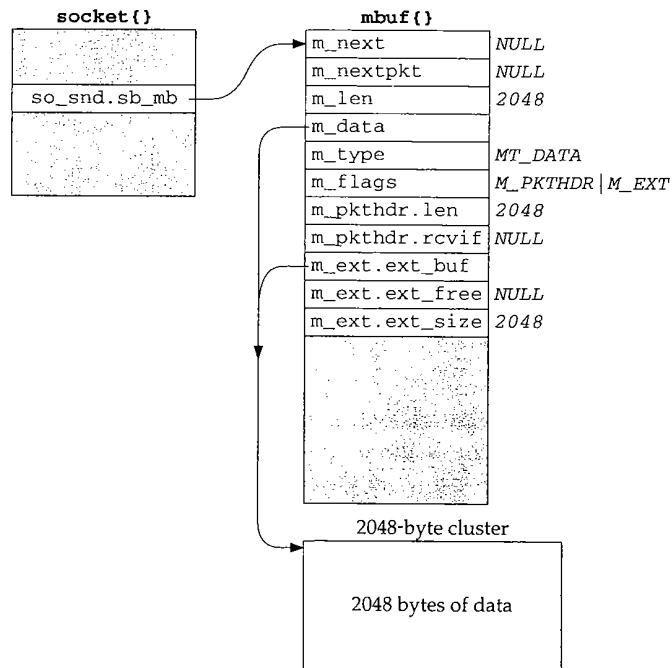


Figure 2.24 TCP socket send buffer containing 2048 bytes of data.

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.

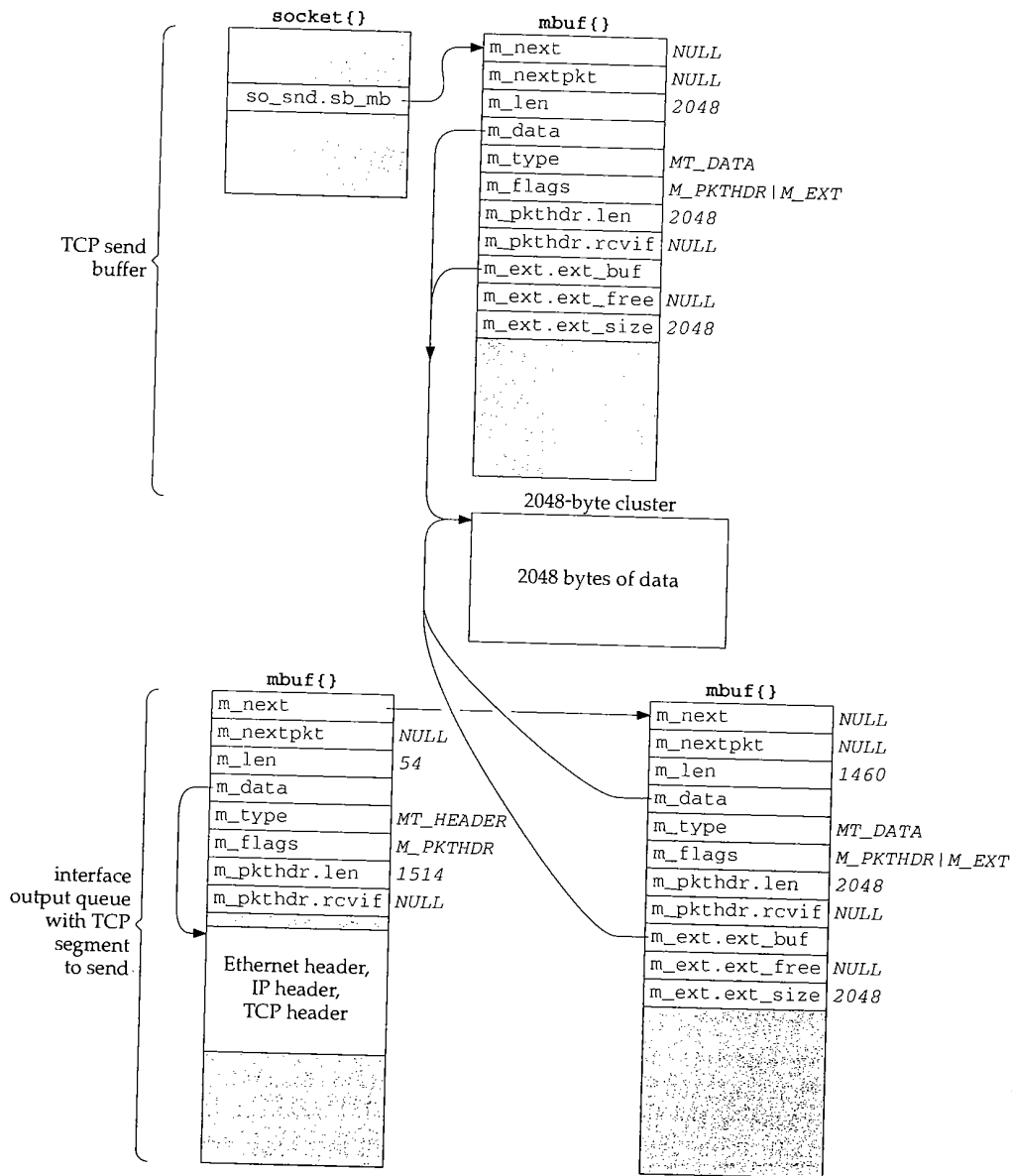


Figure 2.25 TCP socket send buffer and resulting segment on interface's output queue.

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).

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.

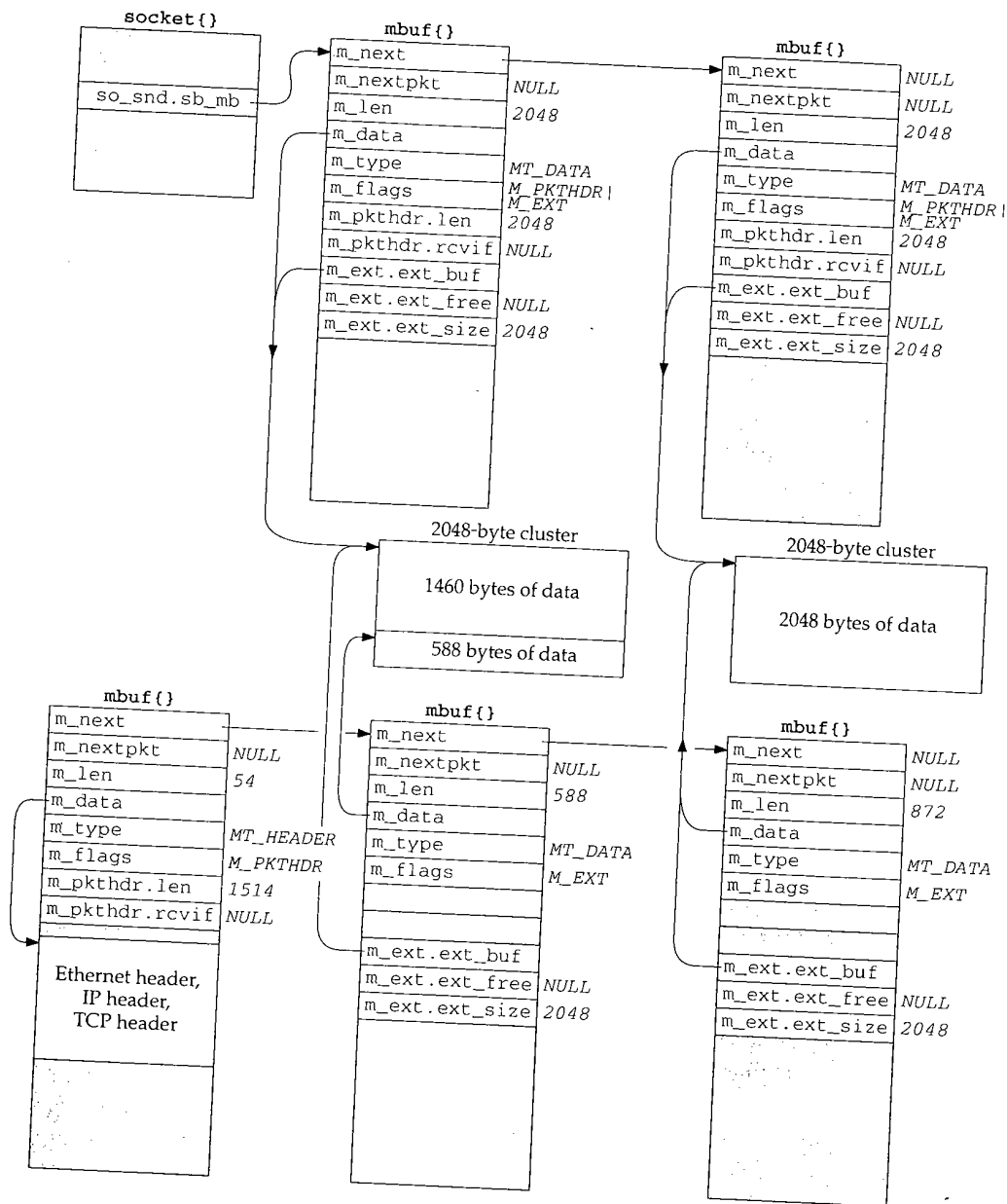


Figure 2.26 Mbuf chain to send next 1460-byte TCP segment.

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 first 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 `so_send` 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.

## 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.

## 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

- 2.1 In Figure 2.9 the `M_COPYFLAGS` value was defined. Why was the `M_EXT` flag not copied?
- 2.2 In Section 2.6 we listed two reasons that `m_pullup` can fail. There are really three reasons. Obtain the source code for this function (Appendix B) and discover the additional reason.
- 2.3 To avoid the problems we described in Section 2.6 with the `dtom` macro when the data is in a cluster, why not just add a back pointer to the mbuf for each cluster?
- 2.4 Since the size of an mbuf cluster is a power of 2 (typically 1024 or 2048), space cannot be taken within the cluster for the reference count. Obtain the Net/3 sources (Appendix B) and determine where these reference counts are stored.
- 2.5 In Figure 2.5 we noted that the two counters `m_drops` and `m_wait` are not currently implemented. Modify the mbuf routines to increment these counters when appropriate.





# 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.
2. A Serial Line IP (SLIP) interface: an example of a point-to-point network running over asynchronous serial lines.

3. A loopback interface: a logical network that returns all outgoing packets as input packets.

### 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.

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

Figure 3.1 Files discussed in this chapter.

#### Global Variables

The global variables introduced in this chapter are described in Figure 3.2.

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)

Figure 3.2 Global variables introduced in this chapter.

#### 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.

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

Figure 3.3 SNMP groups in MIB-II.

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 ISODE package includes a sample SNMP agent for Net/3. See Appendix B for information about ISODE.

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.

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.

Figure 3.4 Simple SNMP variable in the interface group.

### 3.3 ifnet Structure

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

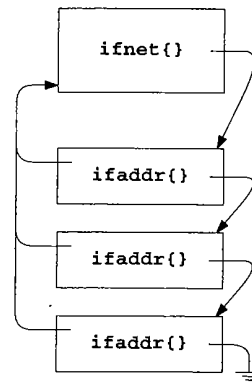


Figure 3.5 Each `ifnet` structure has a list of associated `ifaddr` structures.

The interface in Figure 3.5 is shown with three protocol addresses stored in `ifaddr` structures. Although some network interfaces, such as SLIP, support only a single protocol, others, such as Ethernet, support multiple protocols and need multiple addresses. For example, a system may use a single Ethernet interface for both Internet and OSI protocols. A type field identifies the contents of each Ethernet frame, and since the Internet and OSI protocols employ different addressing schemes, the Ethernet interface must have an Internet address and an OSI address. All the addresses are connected by a linked list (the arrows on the right of Figure 3.5), and each contains a back pointer to the related `ifnet` structure (the arrows on the left of Figure 3.5).

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

This feature first appeared in Net/2. Having two IP addresses for an interface is useful when renumbering a network. During a transition period, the interface can accept packets addressed to the old and new addresses.

The `ifnet` structure is large so we describe it in five sections:

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

Figure 3.6 shows the implementation information contained in the `ifnet` structure.

80-82 `if_next` joins the `ifnet` structures for all the interfaces into a linked list. The `if_attach` function constructs the list during system initialization. `if_addrlist` points to the list of `ifaddr` structures for the interface (Figure 3.16). Each `ifaddr` structure holds addressing information for a protocol that expects to communicate through the interface.

```

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 */

```

Figure 3.6 ifnet structure: implementation information.

### Common interface information

83-86 `if_name` is a short string that identifies the interface type, and `if_unit` identifies multiple instances of the same type. For example, if a system had two SLIP interfaces, both would have an `if_name` consisting of the 2 bytes "sl" and an `if_unit` of 0 for the first interface and 1 for the second. `if_index` uniquely identifies the interface within the kernel and is used by the `sysctl` system call (Section 19.14) as well as in the routing domain.

Sometimes an interface is not uniquely identified by a protocol address. For example, several SLIP connections can have the same local IP address. In these cases, `if_index` specifies the interface explicitly.

`if_flags` specifies the operational state and properties of the interface. A process can examine all the flags but cannot change the flags marked in the "Kernel only" column in Figure 3.7. The flags are accessed with the `SIOCGIFFLAGS` and `SIOCSEIFFLAGS` commands described in Section 4.4.

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

Figure 3.7 if\_flags values.

The IFF\_BROADCAST and IFF\_POINTOPOINT flags are mutually exclusive.

The macro IFF\_CANTCHANGE is a bitwise OR of all the flags in the "Kernel only" column.

The device-specific flags (IFF\_LINKX) may or may not be modifiable by a process depending on the device. For example, Figure 3.29 shows how these flags are defined by the SLIP driver.

### Interface timer

87 `if_timer` is the time in seconds until the kernel calls the `if_watchdog` function for the interface. This function may be used by the device driver to collect interface statistics at regular intervals or to reset hardware that isn't operating correctly.

### BSD Packet Filter

88-89 The next two members, `if_pcount` and `if_bpf`, support the *BSD Packet Filter* (BPF). Through BPF, a process can receive copies of packets transmitted or received by an interface. As we discuss the device drivers, we also describe how packets are passed to BPF. BPF itself is described in Chapter 31.

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

```

90  struct if_data {
91  /* generic interface information */
92      u_char  ifi_type;      /* Figure 3.9 */
93      u_char  ifi_addrln;   /* 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 */
98
99      /* other ifnet members */
100
101
102
103
104
105
106
107
108 #define if_mtu      if_data.ifi_mtu
109 #define if_type     if_data.ifi_type
110 #define if_addrln   if_data.ifi_addrln
111 #define if_hdrlen   if_data.ifi_hdrlen
112 #define if_metric   if_data.ifi_metric
113 #define if_baudrate if_data.ifi_baudrate

```

Figure 3.8 `ifnet` structure: interface characteristics.

Net/3 and this text use the short names provided by the `#define` statements on lines 138 through 143 to specify the `ifnet` members.

### Interface characteristics

90-92 `if_type` specifies the hardware address type supported by the interface. Figure 3.9 lists several common values from `net/if_types.h`.

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

Figure 3.9 *if\_type*: data-link types.

<sup>93-94</sup> *if\_addrLen* is the length of the datalink address and *if\_hdrLen* is the length of the header attached to any outgoing packet by the hardware. An Ethernet network, for example, has an address length of 6 bytes and a header length of 14 bytes (Figure 4.8).

<sup>95</sup> *if\_mtu* is the maximum transmission unit of the interface: the size in bytes of the largest unit of data that the interface can transmit in a single output operation. This is an important parameter that controls the size of packets created by the network and transport protocols. For Ethernet, the value is 1500.

<sup>96-97</sup> *if\_metric* is usually 0; a higher value makes routes through the interface less favorable. *if\_baudrate* specifies the transmission speed of the interface. It is set only by the SLIP interface.

Interface statistics are collected by the next group of members in the *ifnet* structure shown in Figure 3.10.

#### Interface statistics

<sup>98-111</sup> Most of these statistics are self-explanatory. *if\_collisions* is incremented when packet transmission is interrupted by another transmission on shared media such as Ethernet. *if\_noproto* counts the number of packets that can't be processed because the protocol is not supported by the system or the interface (e.g., an OSI packet that arrives at a system that supports only IP). The SLIP interface increments *if\_noproto* if a non-IP packet is placed on its output queue.

These statistics were not part of the *ifnet* structure in Net/1. They were added to support the standard SNMP MIB-II variables for interfaces.

*if\_iqdrops* is accessed only by the SLIP device driver. SLIP and the other network drivers increment *if\_snd.ifq\_drops* (Figure 3.13) when *IF\_DROP* is called. *ifq\_drops* was already in the BSD software when the SNMP statistics were added. The ISODE SNMP agent ignores *if\_iqdrops* and uses *if\_snd.ifq\_drops*.

#### Change timestamp

<sup>112-113</sup> *if\_lastchange* records the last time any of the statistics were changed.

```

98 /* volatile statistics */
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
109                          interface */
110     u_long  ifi_noproto; /* #packets destined for unsupported
111                          protocol */
112     struct timeval ifi_lastchange; /* last updated */
113 } if_data;

/* 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

```

Figure 3.10 ifnet structure: interface statistics.

Once again, Net/3 and this text use the short names provided by the #define statements on lines 144 through 155 to specify the ifnet members.

The next section of the ifnet structure, shown in Figure 3.11, contains pointers to the standard interface-layer functions, which isolate device-specific details from the network layer. Each network interface implements these functions as appropriate for the particular device.

#### Interface functions

114-129 Each device driver initializes its own ifnet structure, including the seven function pointers, at system initialization time. Figure 3.12 describes the generic functions.

We will see the comment /\* XXX \*/ throughout Net/3. It is a warning to the reader that the code is obscure, contains nonobvious side effects, or is a quick solution to a more difficult problem. In this case, it indicates that if\_done is not used in Net/3.



```

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 rentry *);
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);

```

Figure 3.11 ifnet structure: interface procedures.

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

Figure 3.12 ifnet structure: function pointers.

In Chapter 4 we look at the device-specific functions for the Ethernet, SLIP, and loopback interfaces, which the kernel calls indirectly through the pointers in the `ifnet` structure. For example, if `ifp` points to an `ifnet` structure,

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

calls the `if_start` function of the device driver associated with the interface.

The remaining member of the `ifnet` structure is the output queue for the interface and is shown in Figure 3.13.

```

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 };

```

Figure 3.13 ifnet structure: the output queue.

130-137 `if_snd` is the queue of outgoing packets for the interface. Each interface has its own `ifnet` structure and therefore its own output queue. `ifq_head` points to the first packet on the queue (the next one to be output), `ifq_tail` points to the last packet on the queue, `if_len` is the number of packets currently on the queue, and `ifq_maxlen` is the maximum number of buffers allowed on the queue. This maximum is set to 50 (from the global integer `ifqmaxlen`, which is initialized at compile time from `IFQ_MAXLEN`) unless the driver changes it. The queue is implemented as a linked list of mbuf chains. `ifq_drops` counts the number of packets discarded because the queue was full. Figure 3.14 lists the macros and functions that access a queue.

Function	Description
IF_QFULL	Is <i>ifq</i> full?  int <b>IF_QFULL</b> (struct ifqueue *ifq);
IF_DROP	IF_DROP only increments the <code>ifq_drops</code> counter associated with <i>ifq</i> . The name is misleading; the <i>caller</i> drops the packet.  void <b>IF_DROP</b> (struct ifqueue *ifq);
IF_ENQUEUE	Add the packet <i>m</i> to the end of the <i>ifq</i> queue. Packets are linked together by <code>m_nextpkt</code> in the mbuf header.  void <b>IF_ENQUEUE</b> (struct ifqueue *ifq, struct mbuf *m);
IF_PREPEND	Insert the packet <i>m</i> at the front of the <i>ifq</i> queue.  void <b>IF_PREPEND</b> (struct ifqueue *ifq, struct mbuf *m);
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 *ifq, struct mbuf *m);
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 *ifq);

Figure 3.14 ifqueue routines.

The first five routines are macros defined in `net/if.h` and the last routine, `if_qflush`, is a function defined in `net/if.c`. The macros often appear in sequences such as:

```
s = splimp();
if (IF_QFULL(inq)) {
    IF_DROP(inq);          /* queue is full, drop new packet */
    m_freem(m);
} else
    IF_ENQUEUE(inq, m); /* there is room, add to end of queue */
splx(s);
```

This code fragment attempts to add a packet to the queue. If the queue is full, `IF_DROP` increments `ifq_drops` and the packet is discarded. Reliable protocols such as TCP

will retransmit discarded packets. Applications using an unreliable protocol such as UDP must detect and handle the retransmission on their own.

Access to the queue is bracketed by `splimp` and `splx` to block network interrupts and to prevent the network interrupt service routines from accessing the queue while it is in an indeterminate state.

`m_freem` is called before `splx` because the mbuf code has a critical section that runs at `splimp`. It would be wasted effort to call `splx` before `m_freem` only to enter another critical section during `m_freem` (Section 2.5).

### 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.

```

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 };

```

Figure 3.15 ifaddr structure.

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.

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

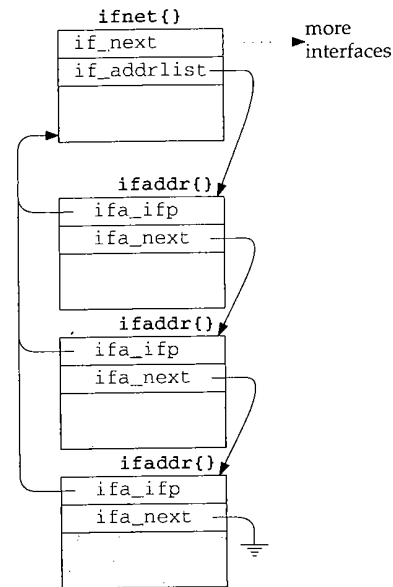


Figure 3.16 ifnet and ifaddr structures.

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 `SIOCIFADDR` 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.

### 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.

```

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 };

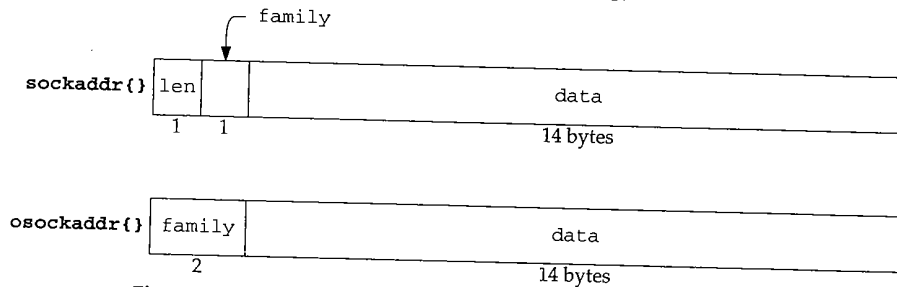
```

socket.h

socket.h

Figure 3.17 sockaddr and osockaddr structures.

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

120-124 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.

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

Figure 3.19 sa\_family constants.

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.

### **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.

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

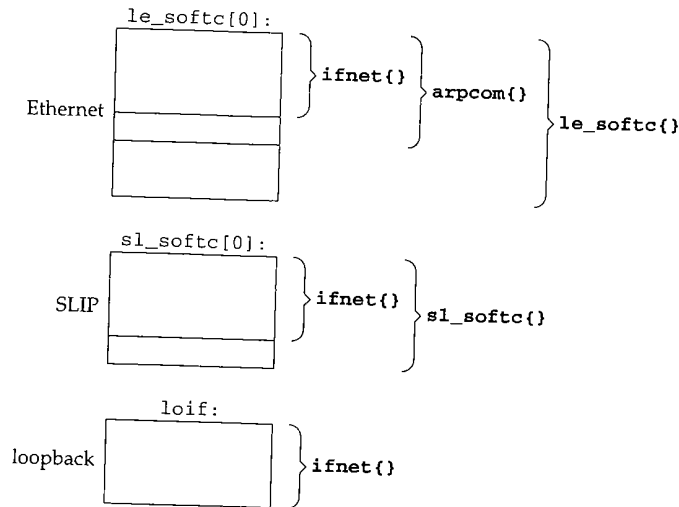


Figure 3.20 Arrangement of `ifnet` structures within device-dependent structures.

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` structures—one 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.

### 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.

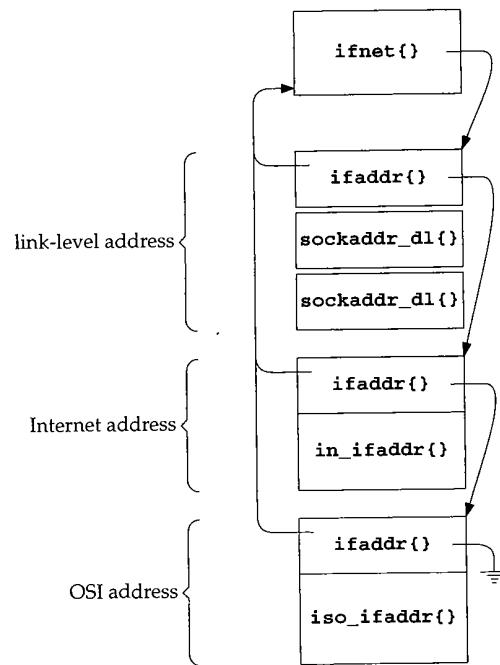


Figure 3.21 An interface address list containing link-level, Internet, and OSI addresses.

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:

```

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

Figure 3.22 `pdevinit` structure.

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.

```

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 }

```

Figure 3.23 main function: network initialization.

- 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.

### 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.

Device	Initialization Function
LANCE Ethernet	leattach
SLIP	slattach
loopback	loopattach

Figure 3.24 Network interface initialization functions.

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).

```

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 */
73
74     /* device-specific members */
75
76 } le_softc[NLE];

```

*if\_le.c*

Figure 3.25 `le_softc` structure.

#### `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.

```

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*

Figure 3.26 `arpcom` structure.

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

106-115 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.

`lestd` 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.

159-162 `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).

```

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;

    /* 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));

    /* 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 = leioclt;
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 }

```

Figure 3.27 leattach function.

### 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.

```

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*

Figure 3.28 `sl_softc` structure.

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.

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

Figure 3.29 SLIP `if_flags` and `sc_flags` values.

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.

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

```

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 = slioc1;
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 }

```

Figure 3.30 slattach function.

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.

### 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.

```

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_addrln = 0;
54     if_attach(ifp);
55     bpfattach(&ifp->if_bpf, ifp, DLT_NULL, sizeof(u_int));
56 }

```

*if\_loop.c*

*if\_loop.c*

Figure 3.31 Loopback interface initialization.

<sup>41-56</sup> 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_addrln` 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.

### 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`.

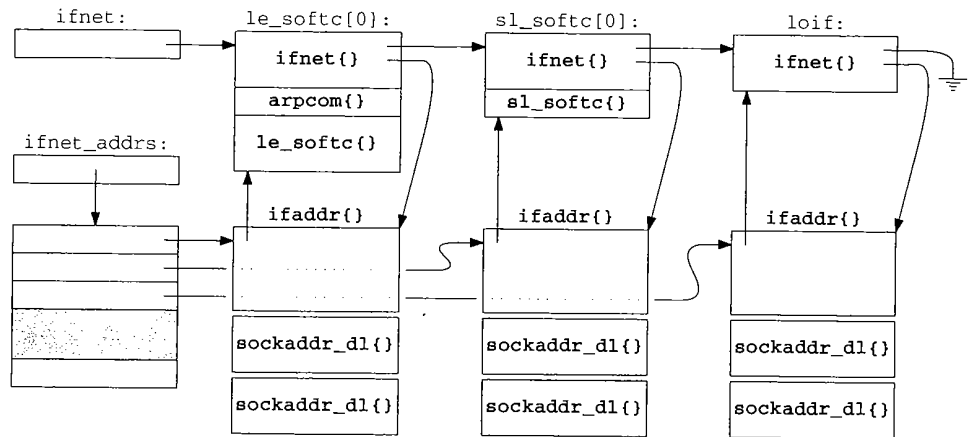


Figure 3.32 ifnet list.

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.

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.



```

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))

```

Figure 3.33 sockaddr\_dl structure.

<sup>61-68</sup> 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 "s10" and the second is called "s11". 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.

<sup>59-74</sup> `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

<sup>75-85</sup> The first time through `if_attach`, the `ifnet_addrs` array doesn't exist so space for 16 entries ( $16 = 8 \ll 1$ ) 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.

```

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_rtrequest();

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     }

```

Figure 3.34 `if_attach` function: assign interface index.

`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.

#### 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

```

86  /* create a Link Level name for this device */
87  unitname = sprintf_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  socksize = masklen + ifp->if_addrlen;
95  #define ROUNDUP(a) (1 + ((a) - 1) | (sizeof(long) - 1))
96  socksize = ROUNDUP(socksize);
97  if (socksize < sizeof(*sdl))
98      socksize = sizeof(*sdl);
99  ifasize = sizeof(*ifa) + 2 * socksize;

```

Figure 3.35 if\_attach function: compute size of link-level address.

(namelen + unitlen). The function rounds socksize, which is masklen plus the hardware address length (if\_addrlen), 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 socksize, 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.

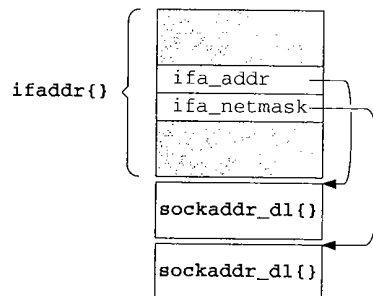


Figure 3.36 The link-level address and mask assigned during if\_attach.

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.

```

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     }

```

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

### 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.

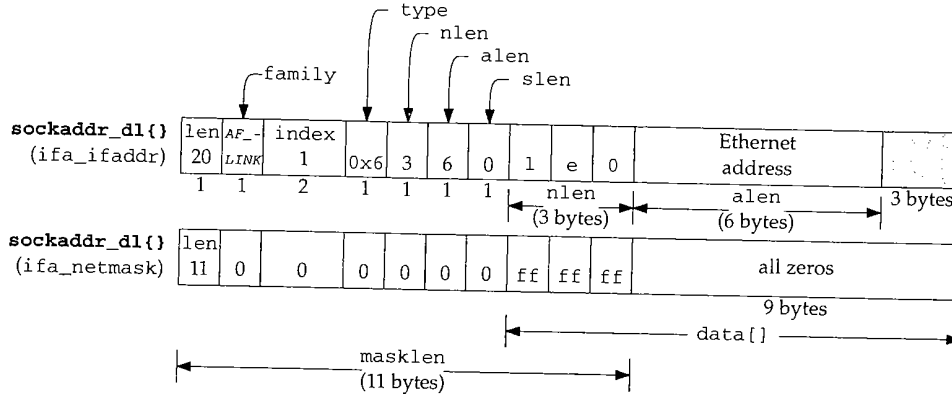


Figure 3.38 The initialized Ethernet `sockaddr_dl` structures (`sdl_prefix` omitted).

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`.

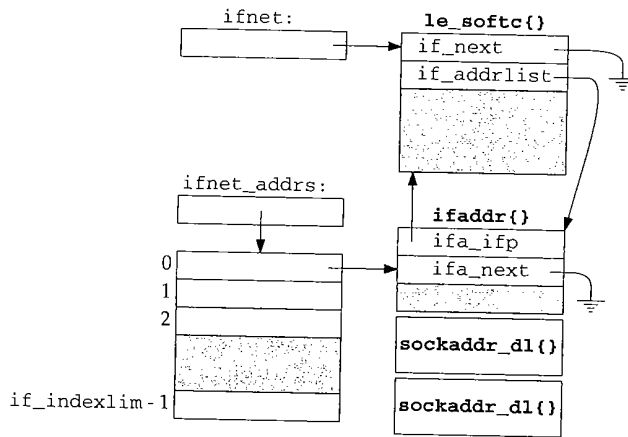


Figure 3.39 The `ifaddr` and `sockaddr_dl` structures after `if_attach` is called for the first time.

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

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.

```

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

```

Figure 3.40 `if_attach` function: Ethernet initialization.**ether\_ifattach function**

The `ether_ifattach` function performs the `ifnet` structure initialization common to all Ethernet devices.

```

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

```

Figure 3.41 `ether_ifattach` function.

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 `arpcmb` structure during system initialization is now copied into the link-level address.

### 3.12 ifinit Function

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

```

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 }

```

Figure 3.42 ifinit function.

<sup>43-51</sup> 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.

```

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

```

Figure 3.43 if\_slowtimo function.

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.

### 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

- 3.1 The `netstat` program on many Unix systems lists network interfaces and their configuration. Try `netstat -i` on a system you have access to. What are the names (`if_name`) and maximum transmission units (`if_mtu`) of the network interfaces?
- 3.2 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?
- 3.3 Why is SLIP's interactive queue shorter than SLIP's standard output queue?
- 3.4 Why aren't `if_hdrlen` and `if_addrlen` initialized in `slattach`?
- 3.5 Draw a picture similar to Figure 3.38 for the SLIP and loopback devices.



# 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.

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.

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.

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
AMD 7990 LANCE Interface	hp300/dev/if_le.c
NE2000 Ethernet	i386/isa/if_ne.c
Western Digital 8003 Ethernet Adapter	i386/isa/if_we.c

Figure 4.1 Ethernet drivers available in Net/3.

ifnet	Ethernet	SLIP	Loopback	Description
if_init	leinit			hardware initialization
if_output	ether_output	sloutput	looutput	accept and queue frame for transmission
if_start	lestart			begin transmission of frame
if_done				output complete (unused)
if_ioctl	leioc1	slioc1	loioc1	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

Figure 4.2 Interface functions for the example drivers.

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 `leioc1`). `if_start` is called only by the `ether_output` function.

## 4.2 Code Introduction

The code for the Ethernet device driver and the generic interface `ioctl`s resides in two headers and three C files, which are listed in Figure 4.3.

File	Description
netinet/if_ether.h	Ethernet structures
net/if.h	ioctl command definitions
net/if_etherubr.c	generic Ethernet functions
hp300/dev/if_le.c	LANCE Ethernet driver
net/if.c	ioctl processing

Figure 4.3 Files discussed in this chapter.

## Global Variables

The global variables shown in Figure 4.4 include the protocol input queues, the LANCE interface structure, and the Ethernet broadcast address.

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

Figure 4.4 Global variables introduced in this chapter.

`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.

ifnet member	Description	Used by SNMP
<code>if_collisions</code>	#collisions on CSMA interfaces	
<code>if_ibytes</code>	total #bytes received	•
<code>if_ierrors</code>	#packets received with input errors	•
<code>if_imcasts</code>	#packets received as multicasts or broadcasts	•
<code>if_ipackets</code>	#packets received on interface	•
<code>if_iqdrops</code>	#packets dropped on input, by this interface	•
<code>if_lastchange</code>	time of last change to statistics	•
<code>if_noproto</code>	#packets destined for unsupported protocol	•
<code>if_obytes</code>	total #bytes sent	•
<code>if_oerrors</code>	#output errors on interface	•
<code>if_omcasts</code>	#packets sent as multicasts	•
<code>if_opackets</code>	#packets sent on interface	•
<code>if_snd.ifq_drops</code>	#packets dropped during output	•
<code>if_snd.ifq_len</code>	#packets in output queue	

Figure 4.5 Statistics maintained in the `ifnet` structure.

Figure 4.6 shows some sample output from the `netstat` command, which includes statistics from the `ifnet` structure.

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, `s10`, `s12`, and `s13` 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,

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

Figure 4.6 Sample interface statistics.

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.

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`.

## 4.3 Ethernet Interface

Net/3 Ethernet device drivers all follow the same general design. This is common for most Unix device drivers because the writer of a driver for a new interface card often starts with a working driver for another card and modifies it. In this section we'll provide a brief overview of the Ethernet standard and outline the design of an Ethernet driver. We'll refer to the LANCE driver to illustrate the design.

Figure 4.8 illustrates Ethernet encapsulation of an IP packet.

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)

Figure 4.7 Variables in interface table: ifTable.

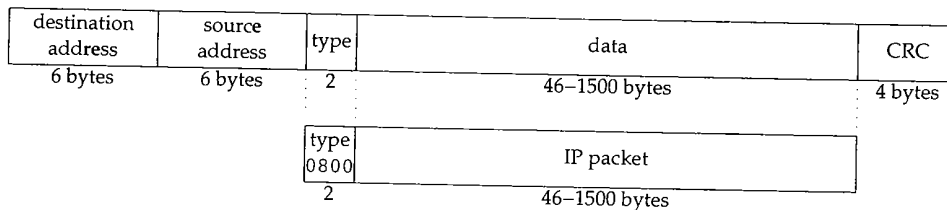


Figure 4.8 Ethernet encapsulation of an IP packet.

Ethernet frames consist of 48-bit destination and source addresses followed by a 16-bit type field that identifies the format of the data carried by the frame. For IP packets, the type is 0x0800 (2048). The frame is terminated with a 32-bit CRC (cyclic redundancy check), which detects errors in the frame.

We are describing the original Ethernet framing standard published in 1982 by Digital Equipment Corp., Intel Corp., and Xerox Corp., as it is the most common form used today in TCP/IP networks. An alternative form is specified by the IEEE (Institute of Electrical and Electronics Engineers) 802.2 and 802.3 standards. Section 2.2 in Volume 1 describes the differences between the two forms. See [Stallings 1987] for more information on the IEEE standards.

Encapsulation of IP packets for Ethernet is specified by RFC 894 [Hornig 1984] and for 802.3 networks by RFC 1042 [Postel and Reynolds 1988].

We will refer to the 48-bit Ethernet addresses as *hardware addresses*. The translation from IP to hardware addresses is done by the ARP protocol described in Chapter 21 (RFC 826 [Plummer 1982]) and from hardware to IP addresses by the RARP protocol (RFC 903 [Finlayson et al. 1984]). Ethernet addresses come in two types, *unicast* and *multicast*. A unicast address specifies a single Ethernet interface, and a multicast address specifies a group of Ethernet interfaces. An Ethernet *broadcast* is a multicast received by all interfaces. Ethernet unicast addresses are assigned by the device's manufacturer, although some devices allow the address to be changed by software.

Some DECNET protocols require the hardware addresses of a multihomed host to be identical, so DECNET must be able to change the Ethernet unicast address of a device.

Figure 4.9 illustrates the data structures and functions that are part of the Ethernet interface.

In figures, a function is identified by an ellipse (`leintr`), data structures by a box (`le_softc[0]`), and a group of functions by a rounded box (ARP protocol).

In the top left corner of Figure 4.9 we show the input queues for the OSI Connectionless Network Layer (`clnl`) protocol, IP, and ARP. We won't say anything more about `clnlintrq`, but include it to emphasize that `ether_input` demultiplexes Ethernet frames into multiple protocol queues.

Technically, OSI uses the term Connectionless Network *Protocol* (CLNP versus CLNL) but we show the terminology used by the Net/3 code. The official standard for CLNP is ISO 8473. [Stallings 1993] summarizes the standard.

The `le_softc` interface structure is in the center of Figure 4.9. We are interested only in the `ifnet` and `arpcom` portions of the structure. The remaining portions are specific to the LANCE hardware. We showed the `ifnet` structure in Figure 3.6 and the `arpcom` structure in Figure 3.26.

### `leintr` Function

We start with the reception of Ethernet frames. For now, we assume that the hardware has been initialized and the system has been configured so that `leintr` is called when the interface generates an interrupt. In normal operation, an Ethernet interface receives frames destined for its unicast hardware address and for the Ethernet broadcast address. When a complete frame is available, the interface generates an interrupt and the kernel calls `leintr`.

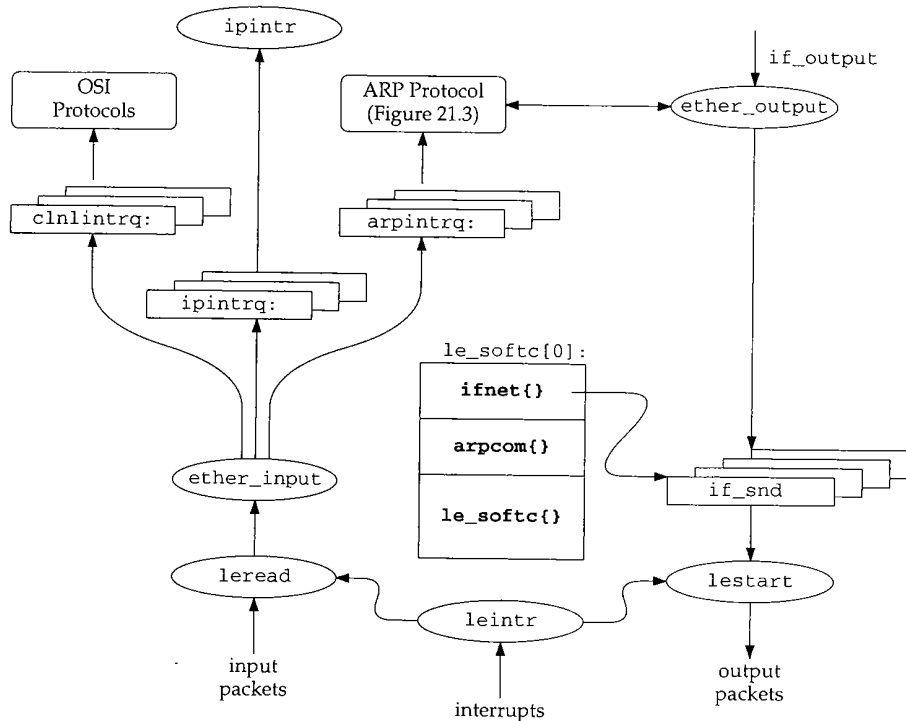


Figure 4.9 Ethernet device driver.

In Chapter 12, we'll see that many Ethernet interfaces may be configured to receive Ethernet multicast frames (other than broadcasts).

Some interfaces can be configured to run in *promiscuous mode* in which the interface receives all frames that appear on the network. The `tcpdump` program described in Volume 1 can take advantage of this feature using BPF.

`leintr` examines the hardware and, if a frame has arrived, calls `leread` to transfer the frame from the interface to a chain of mbufs (with `m_devget`). If the hardware reports that a frame transmission has completed or an error has been detected (such as a bad checksum), `leintr` updates the appropriate interface statistics, resets the hardware, and calls `lestart`, which attempts to transmit another frame.

All Ethernet device drivers deliver their received frames to `ether_input` for further processing. The mbuf chain constructed by the device driver does not include the Ethernet header, so it is passed as a separate argument to `ether_input`. The `ether_header` structure is shown in Figure 4.10.

<sup>38-42</sup> The Ethernet CRC is not generally available. It is computed and checked by the interface hardware, which discards frames that arrive with an invalid CRC. The Ethernet device driver is responsible for converting `ether_type` between network and host byte order. Outside of the driver, it is always in host byte order.

```

-----if_ether.h
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

```

Figure 4.10 The ether\_header structure.

### leread Function

The `leread` function (Figure 4.11) starts with a contiguous buffer of memory passed to it by `leintr` and constructs an `ether_header` structure and a chain of mbufs. The chain contains the data from the Ethernet frame. `leread` also passes the incoming frame to BPF.

```

-----if_le.c
528 leread(unit, buf, len)
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 (bcmp((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             bcmp(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 }

```

if\_le.c

Figure 4.11 leread function.

528-539 The `leintr` function passes three arguments to `leread`: `unit`, which identifies the particular interface card that received a frame; `buf`, which points to the received frame; and `len`, the number of bytes in the frame (including the header and the CRC).

The function constructs the `ether_header` structure by pointing `et` to the front of the buffer and converting the Ethernet type value to host byte order.

540-551 The number of data bytes is computed by subtracting the sizes of the Ethernet header and the CRC from `len`. *Runt packets*, which are too short to be a valid Ethernet frame, are logged, counted, and discarded.

552-557 Next, the destination address is examined to determine if it is the Ethernet broadcast or an Ethernet multicast address. The Ethernet broadcast address is a special case of an Ethernet multicast address; it has every bit set. `etherbroadcastaddr` is an array defined as

```
u_char etherbroadcastaddr[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
```

This is a convenient way to define a 48-bit value in C. This technique works only if we assume that characters are 8-bit values—something that isn't guaranteed by ANSI C.

If `bcmp` reports that `etherbroadcastaddr` and `ether_dhost` are the same, the `M_BCAST` flag is set.

An Ethernet multicast addresses is identified by the low-order bit of the most significant byte of the address. Figure 4.12 illustrates this.

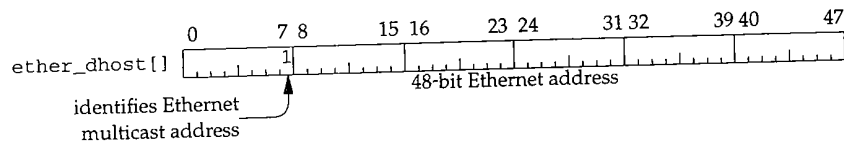


Figure 4.12 Testing for an Ethernet multicast address.

In Chapter 12 we'll see that not all Ethernet multicast frames are IP multicast datagrams and that IP must examine the packet further.

If the multicast bit is on in the address, `M_MCAST` is set in the `flags` variable. The order of the tests is important: first `ether_input` compares the entire 48-bit address to the Ethernet broadcast address, and if they are different it checks the low-order bit of the most significant byte to identify an Ethernet multicast address (Exercise 4.1).

558-573 If the interface is tapped by BPF, the frame is passed directly to BPF by calling `bpf_tap`. We'll see that for SLIP and the loopback interfaces, a special BPF frame is constructed since those networks do not have a link-level header (unlike Ethernet).

When an interface is tapped by BPF, it can be configured to run in promiscuous mode and receive all Ethernet frames that appear on the network instead of the subset of frames normally received by the hardware. The packet is discarded by `lread` if it was sent to a unicast address that does not match the interface's address.

574-585 `m_devget` (Section 2.6) copies the data from the buffer passed to `lread` to an mbuf chain it allocates. The first argument to `m_devget` points to the first byte after the Ethernet header, which is the first data byte in the frame. If `m_devget` runs out of memory, `lread` returns immediately. Otherwise the broadcast and multicast flags are set in the first mbuf in the chain, and `ether_input` processes the packet.

### ether\_input Function

`ether_input`, shown in Figure 4.13, examines the `ether_header` structure to determine the type of data that has been received and then queues the received packet for processing.

```

-----if_ethersubr.c
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 (bcmp((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     switch (eh->ether_type) {
220     case ETHERTYPE_IP:
221         schednetisr(NETISR_IP);
222         inq = &ipintrq;
223         break;

224     case ETHERTYPE_ARP:
225         schednetisr(NETISR_ARP);
226         inq = &arpintrq;
227         break;

228     default:
229         if (eh->ether_type > ETHERMTU) {
230             m_freem(m);
231             return;
232         }

                /* OSI code */

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*

Figure 4.13 ether\_input function.

**Broadcast and multicast recognition**

196-209 The arguments to `ether_input` are `ifp`, a pointer to the receiving interface's `ifnet` structure; `eh`, a pointer to the Ethernet header of the received packet; and `m`, a pointer to the received packet (excluding the Ethernet header).

Any packets that arrive on an inoperative interface are silently discarded. The interface may not have been configured with a protocol address, or may have been disabled by an explicit request from the `ifconfig(8)` program (Section 6.6).

210-218 The variable `time` is a global `timeval` structure that the kernel maintains with the current time and date, as the number of seconds and microseconds past the Unix Epoch (00:00:00 January 1, 1970, Coordinated Universal Time [UTC]). A brief discussion of

UTC can be found in [Itano and Ramsey 1993]. We'll encounter the `timeval` structure throughout the Net/3 sources:

```
struct timeval {
    long tv_sec;      /* seconds */
    long tv_usec;    /* and microseconds */
};
```

`ether_input` updates `if_lastchange` with the current time and increments `if_ibytes` by the size of the incoming packet (the packet length plus the 14-byte Ethernet header).

Next, `ether_input` repeats the tests done by `leread` to determine if the packet is a broadcast or multicast packet.

Some kernels may not have been compiled with the BPF code, so the test must also be done in `ether_input`.

#### Link-level demultiplexing

219-227 `ether_input` jumps according to the Ethernet type field. For an IP packet, `schednetisr` schedules an IP software interrupt and the IP input queue, `ipintrq`, is selected. For an ARP packet, the ARP software interrupt is scheduled and `arpintrq` is selected.

An *isr* is an interrupt service routine.

In previous BSD releases, ARP packets were processed immediately while at the network interrupt level by calling `arpinput` directly. By queueing the packets, they can be processed at the software interrupt level.

If other Ethernet types are to be handled, a kernel programmer would add additional cases here. Alternately, a process can receive other Ethernet types using BPF. For example, RARP servers are normally implemented using BPF under Net/3.

228-307 The default case processes unrecognized Ethernet types or packets that are encapsulated according to the 802.3 standard (such as the OSI connectionless transport). The Ethernet *type* field and the 802.3 *length* field occupy the same position in an Ethernet frame. The two encapsulations can be distinguished because the range of types in an Ethernet encapsulation is distinct from the range of lengths in the 802.3 encapsulation (Figure 4.14). We have omitted the OSI code. [Stallings 1993] contains a description of the OSI link-level protocols.

Range	Description
0 — 1500	IEEE 802.3 <i>length</i> field
1501 — 65535	Ethernet <i>type</i> field:
2048	IP packet
2054	ARP packet

Figure 4.14 Ethernet *type* and 802.3 *length* fields.

There are many additional Ethernet type values that are assigned to various protocols; we don't show them in Figure 4.14. RFC 1700 [Reynolds and Postel 1994] contains a list of the more common types.

### Queue the packet

308-315 Finally, `ether_input` places the packet on the selected queue or discards the packet if the queue is full. We'll see in Figures 7.23 and 21.16 that the default limit for the IP and ARP input queues is 50 (`ipqmaxlen`) packets each.

When `ether_input` returns, the device driver tells the hardware that it is ready to receive the next packet, which may already be present in the device. The packet input queues are processed when the software interrupt scheduled by `schednet_isr` occurs (Section 1.12). Specifically, `ipintr` is called to process the packets on the IP input queue, and `arpintr` is called to process the packets on the ARP input queue.

### ether\_output Function

We now examine the output of Ethernet frames, which starts when a network-level protocol such as IP calls the `if_output` function, specified in the interface's `ifnet` structure. The `if_output` function for all Ethernet devices is `ether_output` (Figure 4.2). `ether_output` takes the data portion of an Ethernet frame, encapsulates it with the 14-byte Ethernet header, and places it on the interface's send queue. This is a large function so we describe it in four parts:

- verification,
- protocol-specific processing,
- frame construction, and
- interface queueing.

Figure 4.15 includes the first part of the function.

49-64 The arguments to `ether_output` are `ifp`, which points to the outgoing interface's `ifnet` structure; `m0`, the packet to send; `dst`, the destination address of the packet; and `rt0`, routing information.

65-67 The macro `senderr` is called throughout `ether_output`.

```
#define senderr(e) ( error = (e); goto bad;)
```

`senderr` saves the error code and jumps to `bad` at the end of the function, where the packet is discarded and `ether_output` returns `error`.

If the interface is up and running, `ether_output` updates the last change time for the interface. Otherwise, it returns `ENETDOWN`.

### Host route

68-74 `rt0` points to the routing entry located by `ip_output` and passed to `ether_output`. If `ether_output` is called from BPF, `rt0` can be null, in which case control passes to the code in Figure 4.16. Otherwise, the route is verified. If the route is not valid, the routing tables are consulted and `EHOSTUNREACH` is returned if a route cannot be located. At this point, `rt0` and `rt` point to a valid route for the next-hop destination.

```

49 int
50 ether_output(ifp, m0, dst, rt0)
51 struct ifnet *ifp;
52 struct mbuf *m0;
53 struct sockaddr *dst;
54 struct rtable *rt0;
55 {
56     short    type;
57     int      s, error = 0;
58     u_char  edst[6];
59     struct mbuf *m = m0;
60     struct rtable *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 = rtallocl(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 = rtallocl(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     }

```

Figure 4.15 ether\_output function: verification.

**Gateway route**

75-85 If the next hop for the packet is a gateway (versus a final destination), a route to the gateway is located and pointed to by `rt`. If a gateway route cannot be found, `EHOSTUNREACH` is returned. At this point, `rt` points to the route for the next-hop destination. The next hop may be a gateway or the final destination.

**Avoid ARP flooding**

86-90 The RTF\_REJECT flag is enabled by the ARP code to discard packets to the destination when the destination is not responding to ARP requests. This is described with Figure 21.24.

ether\_output processing continues according to the destination address of the packet. Since Ethernet devices respond only to Ethernet addresses, to send a packet, ether\_output must find the Ethernet address that corresponds to the IP address of the next-hop destination. The ARP protocol (Chapter 21) implements this translation. Figure 4.16 shows how the driver accesses the ARP protocol.

```

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
103        /* OSI code */
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
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*

*if\_ethersubr.c*

Figure 4.16 ether\_output function: network protocol processing.

**IP output**

91-101 ether\_output jumps according to sa\_family in the destination address. We show only the AF\_INET, AF\_ISO, and AF\_UNSPEC cases in Figure 4.16 and have omitted the code for AF\_ISO.

The AF\_INET case calls arpresolve to determine the Ethernet address corresponding to the destination IP address. If the Ethernet address is already in the ARP cache, arpresolve returns 1 and ether\_output proceeds. Otherwise this IP packet is held by ARP, and when ARP determines the address, it calls ether\_output from the function in\_arpinput.

Assuming the ARP cache contains the hardware address, `ether_output` checks if the packet is going to be broadcast and if the interface is simplex (i.e., it can't receive its own transmissions). If both tests are true, `m_copy` makes a copy of the packet. After the `switch`, the copy is queued as if it had arrived on the Ethernet interface. This is required by the definition of broadcasting; the sending host must receive a copy of the packet.

We'll see in Chapter 12 that multicast packets may also be looped back to be received on the output interface.

#### Explicit Ethernet output

142-146 Some protocols, such as ARP, need to specify the Ethernet destination and type explicitly. The address family constant `AF_UNSPEC` indicates that `dst` points to an Ethernet header. `bcopy` duplicates the destination address in `edst` and assigns the Ethernet type to `type`. It isn't necessary to call `arpresolve` (as for `AF_INET`) because the Ethernet destination address has been provided explicitly by the caller.

#### Unrecognized address families

147-151 Unrecognized address families generate a console message and `ether_output` returns `EAFNOSUPPORT`.

In the next section of `ether_output`, shown in Figure 4.17, the Ethernet frame is constructed.

```

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));

```

Figure 4.17 `ether_output` function: Ethernet frame construction.

#### Ethernet header

152-167 If the code in the `switch` made a copy of the packet, the copy is processed as if it had been received on the output interface by calling `looutput`. The loopback interface and `looutput` are described in Section 5.4.



`M_PREPEND` ensures that there is room for 14 bytes at the front of the packet.

Most protocols arrange to leave room at the front of the mbuf chain so that `M_PREPEND` needs only to adjust some pointers (e.g., `sosend` for UDP output in Section 16.7 and `igmp_sendreport` in Section 13.6).

`ether_output` forms the Ethernet header from `type`, `edst`, and `ac_enaddr` (Figure 3.26). `ac_enaddr` is the unicast Ethernet address associated with the output interface and is the source Ethernet address for all frames transmitted on the interface. `ether_output` overwrites the source address the caller may have specified in the `ether_header` structure with `ac_enaddr`. This makes it more difficult to forge the source address of an Ethernet frame.

At this point, the mbuf contains a complete Ethernet frame except for the 32-bit CRC, which is computed by the Ethernet hardware during transmission. The code shown in Figure 4.18 queues the frame for transmission by the device.

```

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_obytes += 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 }

```

*if\_ethersubr.c*

*if\_ethersubr.c*

Figure 4.18 `ether_output` function: output queueing.

168-185 If the output queue is full, `ether_output` discards the frame and returns `ENOBUFS`. If the output queue is not full, the frame is placed on the interface's send queue, and the interface's `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 and an error code is returned.

## lestart Function

The `lestart` function dequeues frames from the interface output queue and arranges for them to be transmitted by the LANCE Ethernet card. If the device is idle, the function is called to begin transmitting frames. An example appears at the end of `ether_output` (Figure 4.18), where `lestart` is called indirectly through the interface's `if_start` function.

If the device is busy, it generates an interrupt when it completes transmission of the current frame. The driver calls `lestart` to dequeue and transmit the next frame. Once started, the protocol layer can queue frames without calling `lestart` since the driver dequeues and transmits frames until the queue is empty.

Figure 4.19 shows the `lestart` function. `lestart` assumes `splimp` has been called to block any device interrupts.

### Interface must be initialized

325-333 If the interface is not initialized, `lestart` returns immediately.

### Dequeue frame from output queue

335-342 If the interface is initialized, the next frame is removed from the queue. If the interface output queue is empty, `lestart` returns.

### Transmit frame and pass to BPF

343-350 `leput` copies the frame in `m` to the hardware buffer pointed to by the first argument to `leput`. If the interface is tapped by BPF, the frame is passed to `bpf_tap`. We have omitted the device-specific code that initiates the transmission of the frame from the hardware buffer.

### Repeat if device is ready for more frames

359 `lestart` stops passing frames to the device when `le->sc_txcnt` equals `LETBUF`. Some Ethernet interfaces can queue more than one outgoing Ethernet frame. For the LANCE driver, `LETBUF` is the number of hardware transmit buffers available to the driver, and `le->sc_txcnt` keeps track of how many of the buffers are in use.

### Mark device as busy

360-362 Finally, `lestart` turns on `IFF_OACTIVE` in the `ifnet` structure to indicate the device is busy transmitting frames.

There is an unfortunate side effect to queueing multiple frames in the device for transmission. According to [Jacobson 1988a], the LANCE chip is able to transmit queued frames with very little delay between frames. Unfortunately, some [broken] Ethernet devices drop the frames because they can't process the incoming data fast enough.

This interacts badly with an application such as NFS that sends large UDP datagrams (often greater than 8192 bytes) that are fragmented by IP and queued in the LANCE device as multiple Ethernet frames. Fragments are lost on the receiving side, resulting in many incomplete datagrams and high delays as NFS retransmits the entire UDP datagram.

Jacobson noted that Sun's LANCE driver only queued one frame at a time, perhaps to avoid this problem.

```
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
333     if ((le->sc_if.if_flags & IFF_RUNNING) == 0)
334         return (0);
335
336     /* device-specific code */
337
338     do {
339
340         /* device-specific code */
341
342         IF_DEQUEUE(&le->sc_if.if_snd, m);
343         if (m == 0)
344             return (0);
345         len = leput(le->sc_r2->ler2_tbuf[le->sc_tmd], m);
346         /*
347          * If bpf is listening on this interface, let it
348          * see the packet before we commit it to the wire.
349          */
350         if (ifp->if_bpf)
351             bpf_tap(ifp->if_bpf, le->sc_r2->ler2_tbuf[le->sc_tmd],
352                 len);
353
354         /* device-specific code */
355
356     } while (++le->sc_txcnt < LETBUF);
357     le->sc_if.if_flags |= IFF_OACTIVE;
358     return (0);
359 }
360
361
362 if_le.c
```

Figure 4.19 lestart function.

## 4.4 ioctl System Call

The `ioctl` system call supports a generic command interface used by a process to access features of a device that aren't supported by the standard system calls. The prototype for `ioctl` is:

```
int ioctl(int fd, unsigned long com, ...);
```

`fd` is a descriptor, usually a device or network connection. Each type of descriptor supports its own set of `ioctl` commands specified by the second argument, `com`. A third argument is shown as "..." in the prototype, since it is a pointer of some type that depends on the `ioctl` command being invoked. If the command is retrieving information, the third argument must point to a buffer large enough to hold the data. In this text, we discuss only the `ioctl` commands applicable to socket descriptors.

The prototype we show for system calls is the one used by a process to issue the system call. We'll see in Chapter 15 that the function within the kernel that implements a system call has a different prototype.

We describe the implementation of the `ioctl` system call in Chapter 17 but we discuss the implementation of individual `ioctl` commands throughout the text.

The first `ioctl` commands we discuss provide access to the network interface structures that we have described. Throughout the text we summarize `ioctl` commands as shown in Figure 4.20.

Command	Third argument	Function	Description
<code>SIOCGIFCONF</code>	<code>struct ifconf *</code>	<code>ifconf</code>	retrieve list of interface configuration
<code>SIOCGIFFLAGS</code>	<code>struct ifreq *</code>	<code>ifioctl</code>	get interface flags
<code>SIOCGIFMETRIC</code>	<code>struct ifreq *</code>	<code>ifioctl</code>	get interface metric
<code>SIOCSIFFLAGS</code>	<code>struct ifreq *</code>	<code>ifioctl</code>	set interface flags
<code>SIOCSIFMETRIC</code>	<code>struct ifreq *</code>	<code>ifioctl</code>	set interface metric

Figure 4.20 Interface `ioctl` commands.

The first column shows the symbolic constant that identifies the `ioctl` command (the second argument, `com`). The second column shows the type of the third argument passed to the `ioctl` system call for the command shown in the first column. The third column names the function that implements the command.

Figure 4.21 shows the organization of the various functions that process `ioctl` commands. The shaded functions are the ones we describe in this chapter. The remaining functions are described in other chapters.

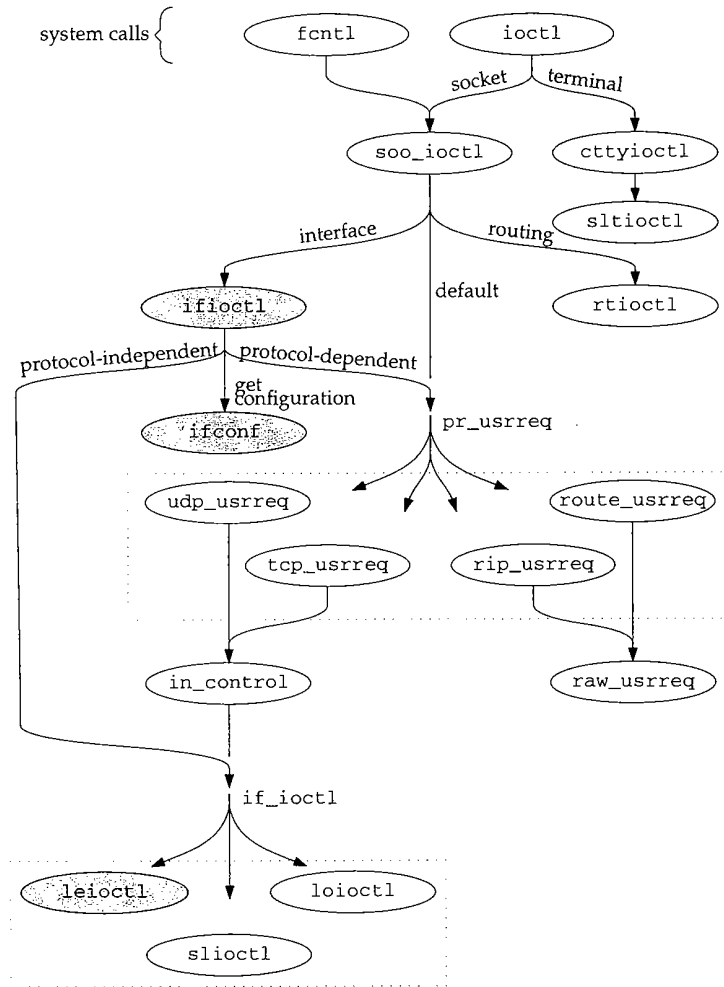


Figure 4.21 ioctl functions described in this chapter.

**ifioctl Function**

The `ioctl` system call routes the five commands shown in Figure 4.20 to the `ifioctl` function shown in Figure 4.22.

```

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) {
411
412         /* other interface ioctl commands (Figures 4.29 and 12.11) */
413
414     default:
415         if (so->so_proto == 0)
416             return (EOPNOTSUPP);
417         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
418             cmd, data, ifp));
419     }
420     return (0);
421 }

```

**Figure 4.22** `ifioctl` function: overview and `SIOCGIFCONF`.

394-405 For the `SIOCGIFCONF` command, `ifioctl` calls `ifconf` to construct a table of variable-length `ifreq` structures.

406-410 For the remaining `ioctl` commands, the `data` argument is a pointer to an `ifreq` structure. `ifunit` searches the `ifnet` list for an interface with the text name provided by the process in `ifr->ifr_name` (e.g., "sl0", "le1", or "lo0"). If there is no matching interface, `ifioctl` returns `ENXIO`. The remaining code depends on `cmd` and is described with Figure 4.29.

447-454 If the interface `ioctl` command is not recognized, `ifioctl` forwards the command to the user-request function of the protocol associated with the socket on which the request was made. For IP, these commands are issued on a UDP socket and `udp_usrreq` is called. The commands that fall into this category are described in Figure 6.10. Section 23.10 describes the `udp_usrreq` function in detail.

If control falls out of the switch, 0 is returned.

**ifconf Function**

`ifconf` provides a standard way for a process to discover the interfaces present and the addresses configured on a system. Interface information is represented by `ifreq` and `ifconf` structures shown in Figures 4.23 and 4.24.

```

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 4.23 `ifreq` structure.

262-279 An `ifreq` structure contains the name of an interface in `ifr_name`. The remaining members in the union are accessed by the various `ioctl` commands. As usual, macros simplify the syntax required to access the members of the union.

```

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

Figure 4.24 `ifconf` structure.

292-300 In the `ifconf` structure, `ifc_len` is the size in bytes of the buffer pointed to by `ifc_buf`. The buffer is allocated by a process but filled in by `ifconf` with an array of variable-length `ifreq` structures. For the `ifconf` function, `ifr_addr` is the relevant member of the union in the `ifreq` structure. Each `ifreq` structure has a variable length because the length of `ifr_addr` (a `sockaddr` structure) varies according to the type of address. The `sa_len` member from the `sockaddr` structure must be used to

locate the end of each entry. Figure 4.25 illustrates the data structures manipulated by `ifconf`.

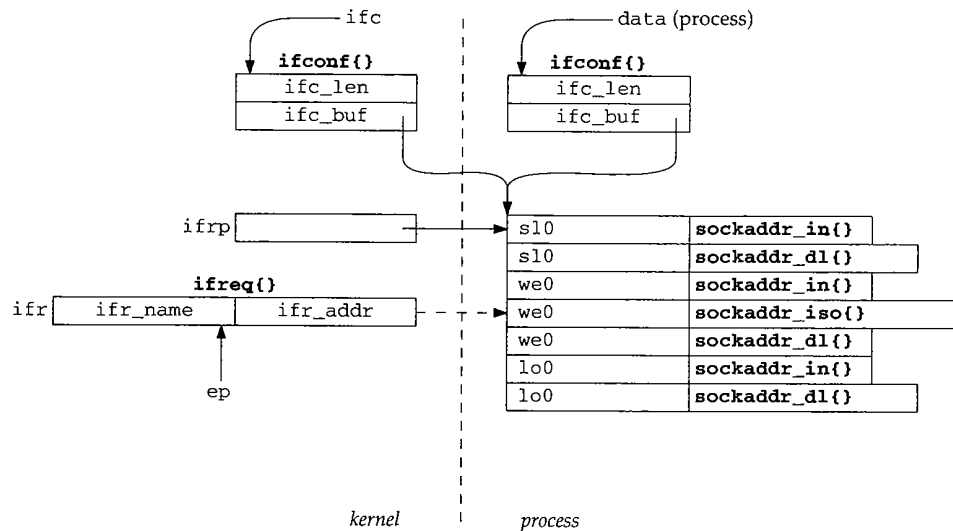


Figure 4.25 `ifconf` data structures.

In Figure 4.25, the data on the left is in the kernel and the data on the right is in a process. We'll refer to this figure as we discuss the `ifconf` function listed in Figure 4.26.

462-474 The two arguments to `ifconf` are: `cmd`, which is ignored; and `data`, which points to a copy of the `ifconf` structure specified by the process.

`ifc` is `data` cast to a `ifconf` structure pointer. `ifrp` traverses the interface list starting at `ifnet` (the head of the list), and `ifa` traverses the address list for each interface. `cp` and `ep` control the construction of the text interface name within `ifr`, which is the `ifreq` structure that holds an interface name and address before they are copied to the process's buffer. `ifrp` points to this buffer and is advanced after each address is copied. `space` is the number of bytes remaining in the process's buffer, `cp` is used to search for the end of the name, and `ep` marks the last possible location for the numeric portion of the interface name.

```

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

Figure 4.26 ifconf function.

475-488 The for loop traverses the list of interfaces. For each interface, the text name is copied to `ifr_name` followed by the text representation of the `if_unit` number. If no addresses have been assigned to the interface, an address of all 0s is constructed, the resulting `ifreq` structure is copied to the process, `space` is decreased, and `ifrp` is advanced.

489-515 If the interface has one or more addresses, the for loop processes each one. The

address is added to the interface name in `ifr` and then `ifr` is copied to the process. Addresses longer than a standard `sockaddr` structure don't fit in `ifr` and are copied directly out to the process. After each address, space and `ifr.p` are adjusted. After all the interfaces are processed, the length of the buffer is updated (`ifc->ifc_len`) and `ifconf` returns. The `ioctl` system call takes care of copying the new contents of the `ifconf` structure back to the `ifconf` structure in the process.

### Example

Figure 4.27 shows the configuration of the interface structures after the Ethernet, SLIP, and loopback interfaces have been initialized.

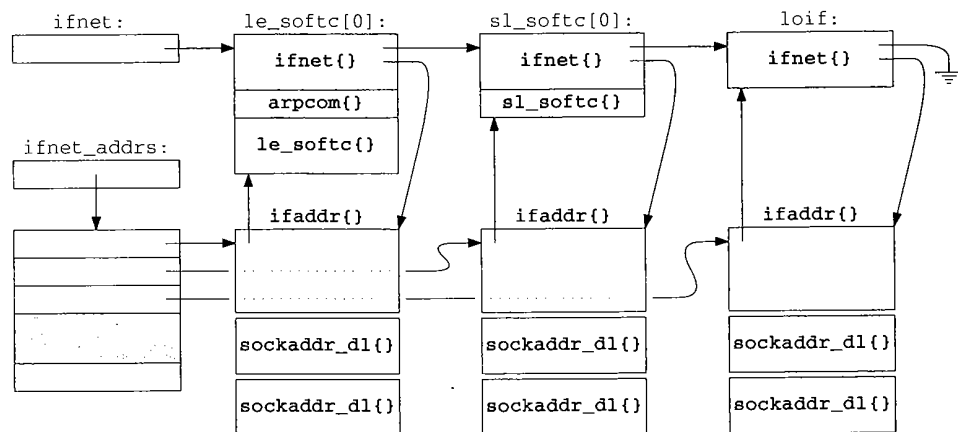


Figure 4.27 Interface and address data structures.

Figure 4.28 shows the contents of `ifc` and buffer after the following code is executed.

```

struct ifconf ifc;    /* SIOCGIFCONF adjusts this */
char buffer[144];    /* contains interface addresses when ioctl returns */
int s;                /* any socket */

ifc.ifc_len = 144;
ifc.ifc_buf = buffer;
if (ioctl(s, SIOCGIFCONF, &ifc) < 0) {
    perror("ioctl failed");
    exit(1);
}

```

There are no restrictions on the type of socket specified with the `SIOCGIFCONF` command, which, as we have seen, returns the addresses for all protocol families.

In Figure 4.28, `ifc_len` has been changed from 144 to 108 by `ioctl` since the three addresses returned in the buffer only occupy 108 ( $3 \times 36$ ) bytes. Three `sockaddr_dl` addresses are returned and the last 36 bytes of the buffer are unused. The first 16 bytes of each entry contain the text name of the interface. In this case only 3 of the 16 bytes are used.

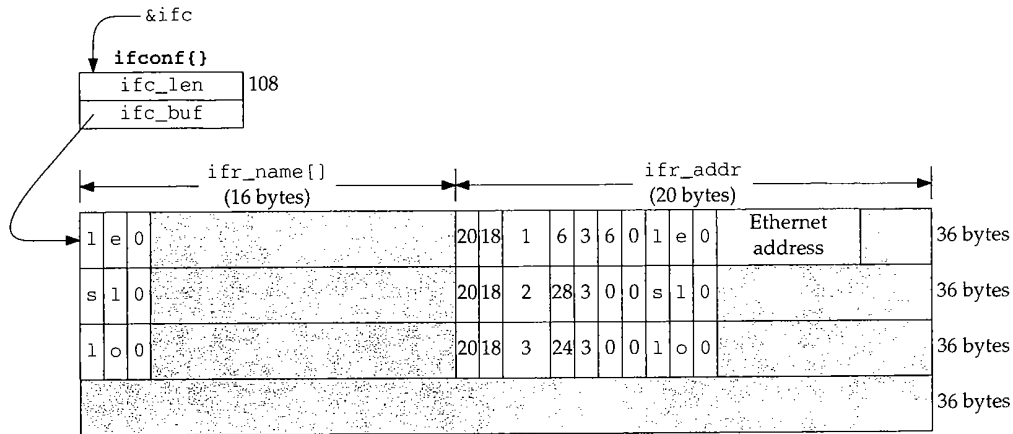


Figure 4.28 Data returned by the SIOCGIFCONF command.

`ifr_addr` has the form of a `sockaddr` structure, so the first value is the length (20 bytes) and the second value is the type of address (18, `AF_LINK`). The next value is `sdl_index`, which is different for each interface as is `sdl_type` (6, 28, and 24 correspond to `IFT_ETHER`, `IFT_SLIP`, and `IFT_LOOP`).

The next three values are `sa_nlen` (the length of the text name), `sa_alen` (the length of the hardware address), and `sa_slens` (unused). `sa_nlen` is 3 for all three entries. `sa_alen` is 6 for the Ethernet address and 0 for both the SLIP and loopback interfaces. `sa_slens` is always 0.

Finally, the text interface name appears, followed by the hardware address (Ethernet only). Neither the SLIP nor the loopback interface store a hardware-level address in the `sockaddr_dl` structure.

In the example, only `sockaddr_dl` addresses are returned (because no other address types were configured in Figure 4.27), so each entry in the buffer is the same size. If other addresses (e.g., IP or OSI addresses) were configured for an interface, they would be returned along with the `sockaddr_dl` addresses, and the size of each entry would vary according to the type of address returned.

### Generic Interface ioctl commands

The four remaining interface commands from Figure 4.20 (`SIOCGIFFLAGS`, `SIOCGIFMETRIC`, `SIOCSIFFLAGS`, and `SIOCSIFMETRIC`) are handled by the `ifioctl` function. Figure 4.29 shows the case statements for these commands.

#### SIOCGIFFLAGS and SIOCGIFMETRIC

410-416 For the two `SIOCGxxx` commands, `ifioctl` copies the `if_flags` or `if_metric` value for the interface into the `ifreq` structure. For the flags, the `ifr_flags` member of the union is used and for the metric, the `ifr_metric` member is used (Figure 4.23).

```

410     switch (cmd) {
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             (void) (*ifp->if_ioctl) (ifp, cmd, data);
436         break;
437
438     case SIOCSIFMETRIC:
439         if (error = suser(p->p_ucred, &p->p_acflag))
440             return (error);
441         ifp->if_metric = ifr->ifr_metric;
442         break;

```

Figure 4.29 ifioctl function: flags and metrics.

**SIOCSIFFLAGS**

417-429 To change the interface flags, the calling process must have superuser privileges. If the process is shutting down a running interface or bringing up an interface that isn't running, `if_down` or `if_up` are called respectively.

**Ignore IFF\_CANTCHANGE flags**

430-434 Recall from Figure 3.7 that some interface flags cannot be changed by a process. The expression `(ifp->if_flags & IFF_CANTCHANGE)` clears the interface flags that *can* be changed by the process, and the expression `(ifr->ifr_flags & ~IFF_CANTCHANGE)` clears the flags in the *request* that may *not* be changed by the process. The two expressions are ORed together and saved as the new value for `ifp->if_flags`. Before returning, the request is passed to the `if_ioctl` function associated with the device (e.g., `lei_ioctl` for the LANCE driver—Figure 4.31).

**SIOCSIFMETRIC**

435-439 Changing the interface metric is easier; as long as the process has superuser privileges, `ifioctl` copies the new metric into `if_metric` for the interface.

**if\_down and if\_up Functions**

With the `ifconfig` program, an administrator can enable and disable an interface by setting or clearing the `IFF_UP` flag through the `SIOCSIFFLAGS` command. Figure 4.30 shows the code for the `if_down` and `if_up` functions.

```

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 }

```

*if.c*

Figure 4.30 `if_down` and `if_up` functions.

292-302 When an interface is shut down, the `IFF_UP` flag is cleared and the `PRC_IFDOWN` command is issued by `pfctlinput` (Section 7.7) for each address associated with the interface. This gives each protocol an opportunity to respond to the interface being shut down. Some protocols, such as OSI, terminate connections using the interface. IP attempts to reroute connections through other interfaces if possible. TCP and UDP ignore failing interfaces and rely on the routing protocols to find alternate paths for the packets.

`if_qflush` discards any packets queued for the interface. The routing system is notified of the change by `rt_ifmsg`. TCP retransmits the lost packets automatically; UDP applications must explicitly detect and respond to this condition on their own.

308-315 When an interface is enabled, the `IFF_UP` flag is set and `rt_ifmsg` notifies the routing system that the interface status has changed.

### Ethernet, SLIP, and Loopback

We saw in Figure 4.29 that for the SIOCSIFFLAGS command, `ifioctl` calls the `if_ioctl` function for the interface. In our three sample interfaces, the `sliioctl` and `loioctl` functions return `EINVAL` for this command, which is ignored by `ifioctl`. Figure 4.31 shows the `leiioctl` function and SIOCSIFFLAGS processing of the LANCE Ethernet driver.

```

614 leiioctl(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 lereg1 *ler1 = le->sc_r1;
622     int s = splimp(), error = 0;
623     switch (cmd) {
624
625         /* SIOCSIFADDR code (Figure 6.28) */
626
627     case SIOCSIFFLAGS:
628         if ((ifp->if_flags & IFF_UP) == 0 &&
629             ifp->if_flags & IFF_RUNNING) {
630             LERDWR(le->sc_r0, LE_STOP, ler1->ler1_rdp);
631             ifp->if_flags &= ~IFF_RUNNING;
632         } else if (ifp->if_flags & IFF_UP &&
633                 (ifp->if_flags & IFF_RUNNING) == 0)
634             leinit(ifp->if_unit);
635         /*
636          * If the state of the promiscuous bit changes, the interface
637          * must be reset to effect the change.
638          */
639         if (((ifp->if_flags ^ le->sc_iflags) & IFF_PROMISC) &&
640             (ifp->if_flags & IFF_RUNNING)) {
641             le->sc_iflags = ifp->if_flags;
642             lereset(ifp->if_unit);
643             lestart(ifp);
644         }
645         break;
646
647         /* SIOCADMULTI and SIOCDELMULTI code (Figure 12.31) */
648
649     default:
650         error = EINVAL;
651     }
652     splx(s);
653     return (error);
654 }

```

Figure 4.31 `leiioctl` function: SIOCSIFFLAGS.

614-623 `leioc1` casts the third argument, `data`, to an `ifaddr` structure pointer and saves the value in `ifa`. The `le` pointer references the `le_softc` structure indexed by `ifp->if_unit`. The switch statement, based on `cmd`, makes up the main body of the function.

638-656 Only the `SIOCSIFFLAGS` case is shown in Figure 4.31. By the time `ifioc1` calls `leioc1`, the interface flags have been changed. The code shown here forces the physical interface into a state that matches the configuration of the flags. If the interface is going down (`IFF_UP` is not set), but the interface is operating, the interface is shut down. If the interface is going up but is not operating, the interface is initialized and restarted.

If the promiscuous bit has been changed, the interface is shut down, reset, and restarted to implement the change.

The expression including the exclusive OR and `IFF_PROMISC` is true only if the request changes the `IFF_PROMISC` bit.

672-677 The default case for unrecognized commands posts `EINVAL`, which is returned at the end of the function.

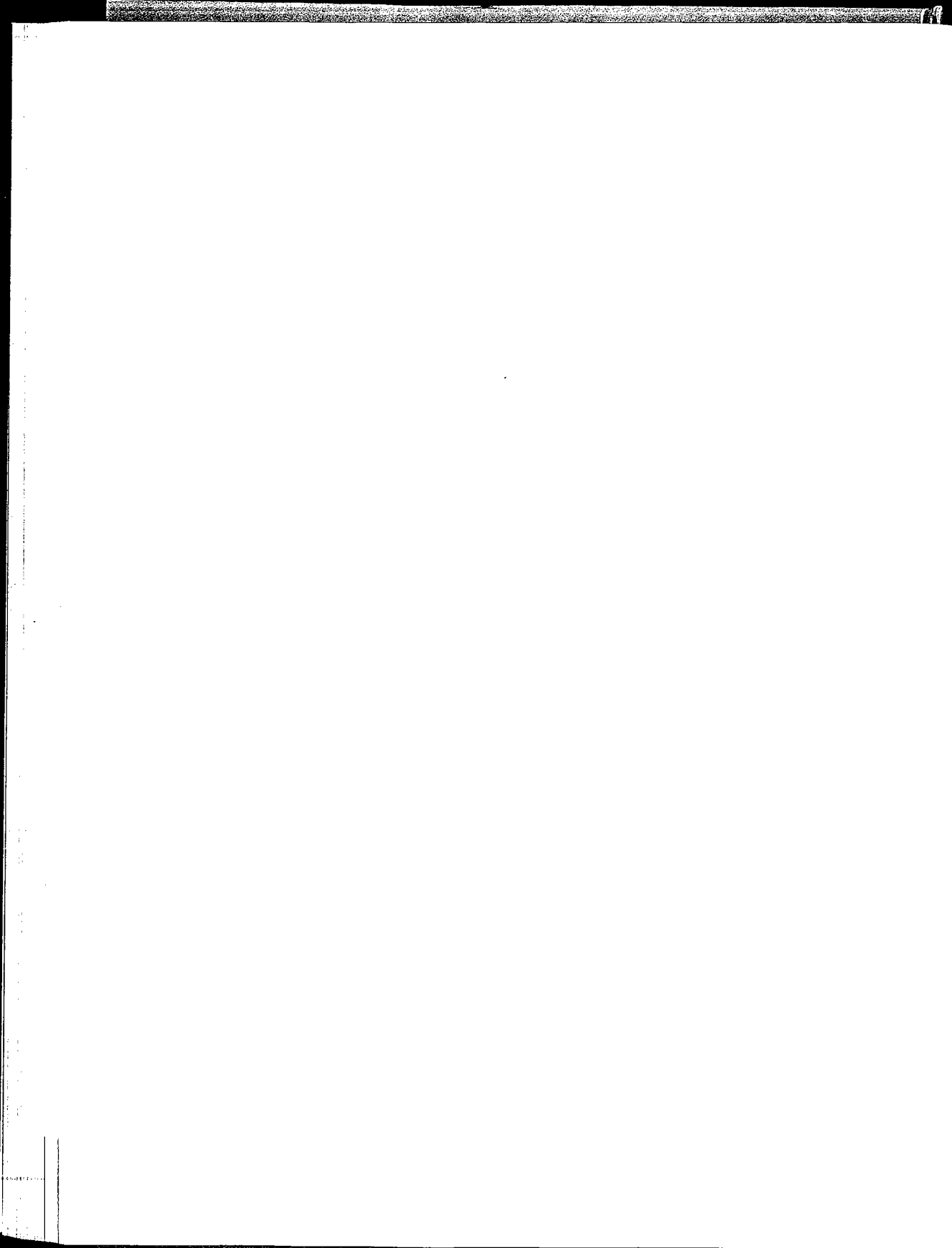
## 4.5 Summary

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 `ioc1` commands that access the interface-layer data structures.

## Exercises

- 4.1 In `lread`, the `M_MCAST` flag (in addition to `M_BCAST`) is always set when a broadcast packet is received. Compare this behavior to the code in `ether_input`. Why are the flags set in `lread` and `ether_input`? Does it matter? Which is correct?
- 4.2 In `ether_input` (Figure 4.13), what would happen if the test for the broadcast address and the test for a multicast address were swapped? What would happen if the `if` on the test for a multicast address were not preceded by an `else`?





# 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.

ifnet	Ethernet	SLIP	Loopback	Description
<code>if_init</code>	<code>leinit</code>			initialize hardware
<code>if_output</code>	<code>ether_output</code>	<code>sloutput</code>	<code>looutput</code>	accept and queue packet for transmission
<code>if_start</code>	<code>lestart</code>			begin transmission of frame
<code>if_done</code>				output complete (unused)
<code>if_ioctl</code>	<code>leioctl</code>	<code>slioclt</code>	<code>loioclt</code>	handle <code>ioctl</code> commands from a process
<code>if_reset</code>	<code>lereset</code>			reset the device to a known state
<code>if_watchdog</code>				watch the device for failures or collect statistics

Figure 5.1 Interface functions for the example drivers.

### 5.2 Code Introduction

The files containing code for SLIP and loopback drivers are listed in Figure 5.2.

File	Description
net/if_slvar.h	SLIP definitions
net/if_sl.c	SLIP driver functions
net/if_loop.c	loopback driver

Figure 5.2 Files discussed in this chapter.

### Global Variables

The SLIP and loopback interface structures are described in this chapter.

Variable	Datatype	Description
sl_softc	struct sl_softc []	SLIP interface
loif	struct ifnet	loopback interface

Figure 5.3 Global variables introduced in this chapter.

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.

Variable	Description	Used by SNMP
tk_nin	#bytes received by any serial interface (updated by SLIP driver)	

Figure 5.4 tk\_nin variable.

## 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.

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.

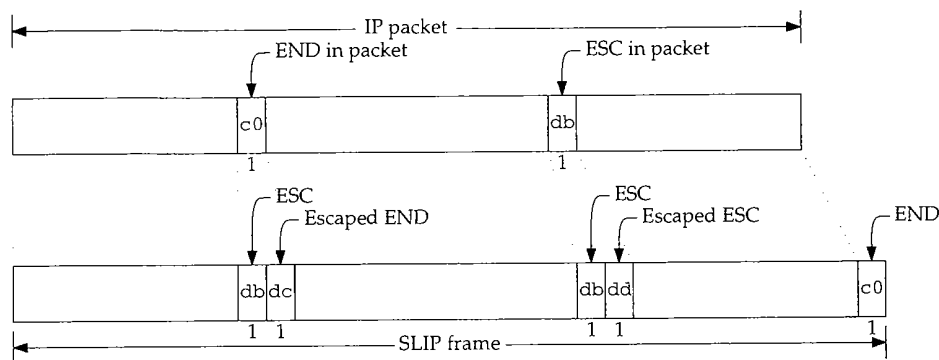


Figure 5.5 SLIP encapsulation of an IP packet.

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.

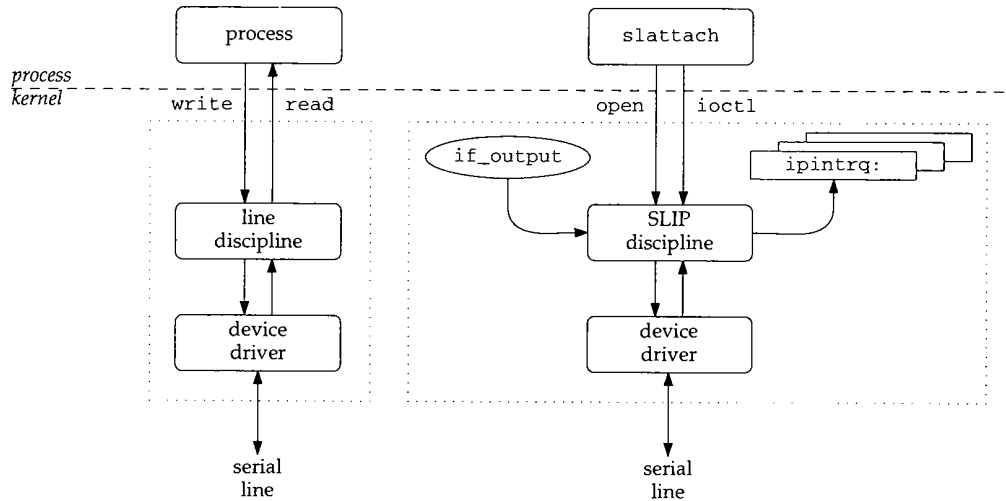


Figure 5.6 The SLIP interface as a line discipline.

Function	Network Interface	Line Discipline	Description
slattach	•		initialize and attach <code>sl_softc</code> structures to <code>ifnet</code> list
slinit	•		initialize the SLIP data structures
sloutput	•		queue outgoing packets for transmission on associated TTY device
slioctl	•		process socket <code>ioctl</code> requests
sl_btom	•		convert a device buffer to an mbuf chain
slopen		•	attach <code>sl_softc</code> structure to TTY device and initialize driver
slclose		•	detach <code>sl_softc</code> structure from TTY device, mark interface as down, and release memory
sltioctl		•	process TTY <code>ioctl</code> 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

Figure 5.7 The functions in the SLIP device driver.

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.

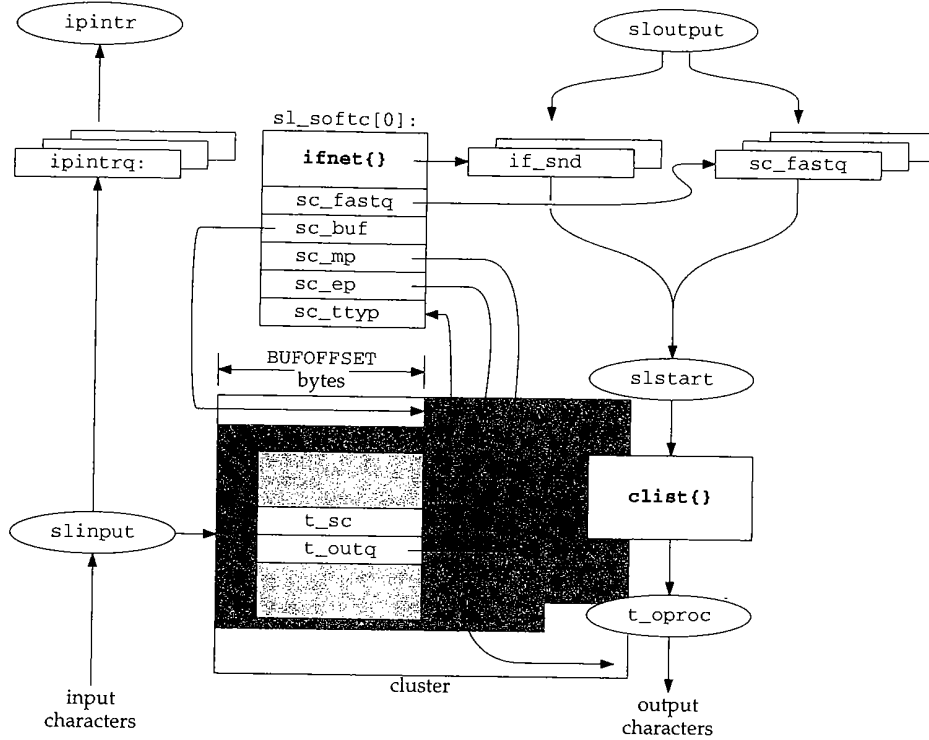


Figure 5.8 SLIP device driver.

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 illustrate 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 `slnit`**

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.

```

181 int
182 slopen(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     if (error = suser(p->p_ucred, &p->p_acflag))
191         return (error);
192     if (tp->t_line == SLIPDISC)
193         return (0);
194     for (nsl = NSL, sc = sl_softc; --nsl >= 0; sc++)
195         if (sc->sc_ttyp == NULL) {
196             if (slnit(sc) == 0)
197                 return (ENOBUFFS);
198             tp->t_sc = (caddr_t) sc;
199             sc->sc_ttyp = tp;
200             sc->sc_if.if_baudrate = tp->t_ospeed;
201             ttyflush(tp, FREAD | FWRITE);
202             return (0);
203         }
204     return (ENXIO);
205 }

```

*if\_sl.c*

*if\_sl.c*

Figure 5.9 The `slopen` function.

181-193 Two arguments are passed to `slopen`: `dev`, a kernel device identifier that `slopen` 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, `slopen` returns immediately.

194-205 The `for` loop searches the array of `sl_softc` structures for the first unused entry, calls `slnit` (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.

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 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 }

```

Figure 5.10 The `slinit` function.

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.

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.

Constant	Value	Description
<i>MCLBYTES</i>	2048	size of an mbuf cluster
<i>SLBUFSIZE</i>	2048	maximum size of an uncompressed SLIP packet—including a BPF header
<i>SLIP_HDRLEN</i>	16	size of SLIP BPF header
<i>BUFOFFSET</i>	148	maximum size of an expanded TCP/IP header plus room for a BPF header
<i>SLMAX</i>	1900	maximum size of a compressed SLIP packet stored in a cluster
<i>SLMTU</i>	296	optimal size of SLIP packet; results in minimal delay with good bulk throughput
<i>SLIP_HIWAT</i>	100	maximum number of bytes to queue in TTY output queue
BUFOFFSET + SLMAX = SLBUFSIZE = MCLBYTES		

Figure 5.11 SLIP constants.

**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.

```

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
538     tk_nin++;
539     sc = (struct sl_softc *) tp->t_sc;
540     if (sc == NULL)
541         return;
542     if (c & TTY_ERRORMASK || ((tp->t_state & TS_CARR_ON) == 0 &&
543         (tp->t_cflag & CLOCAL) == 0)) {
544         sc->sc_flags |= SC_ERROR;
545         return;
546     }
547     c &= TTY_CHARMASK;
548
549     ++sc->sc_if.if_ibytes;
550
551     switch (c) {
552     case TRANS_FRAME_ESCAPE:
553         if (sc->sc_escape)
554             c = FRAME_ESCAPE;
555         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:

        /* FRAME_END code (Figure 5.13) */

636     }
637     if (sc->sc_mp < sc->sc_ep) {
638         *sc->sc_mp++ = c;
639         sc->sc_escape = 0;
640         return;
641     }
642     /* can't put lower; would miss an extra frame */
643     sc->sc_flags |= SC_ERROR;

644     error:
645     sc->sc_if.if_ierrors++;
646     newpack:
647     sc->sc_mp = sc->sc_buf = sc->sc_ep - SLMAX;
648     sc->sc_escape = 0;
649 }

```

if\_sl.c

Figure 5.12 slinput function.

527-545 The arguments to `slinput` 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. `slinput` converts `tp->t_sc` to `sc`, a pointer to an `sl_softc` structure. If there is no interface associated with the TTY device, `slinput` returns immediately.

The first argument to `slinput` 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 *slinput* 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.

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     case FRAME_END:
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_UNCOMPRESSEDED_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.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*

Figure 5.13 slinput function: end-of-frame processing.

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`.

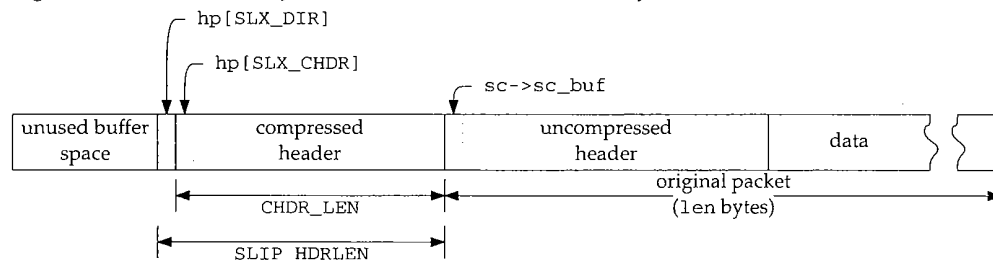


Figure 5.14 SLIP packet in BPF format.

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, `slinput` 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 `slinput` jumps to `newpack`, where it updates the cluster packet pointers and clears `sc_escape`.

While the SLIP driver increments `if_errors` 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 `slinput` 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).

```

-----if_sl.c
259 int
260 sloutput(ifp, m, dst, rtp)
261 struct ifnet *ifp;
262 struct mbuf *m;
263 struct sockaddr *dst;
264 struct rtpentry *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 slioctrl). 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_freem(m);
278         sc->sc_if.if_noproto++;
279         return (EAFNOSUPPORT);
280     }
281     if (sc->sc_ttyp == NULL) {
282         m_freem(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_freem(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_freem(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_freem(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*

Figure 5.15 sloutput function.

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 queuing 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) `sloput` 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`), `sloput` 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 `sloput`, 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.

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, `sloput` 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.

```
if_sl.c
318 void
319 slstart(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     for (;;) {
332         /*
333          * If there is more in the output queue, just send it now.
334          * We are being called in lieu of ttstart and must do what
335          * it would.
336          */
337         if (tp->t_outq.c_cc != 0) {
338             (*tp->t_oprof) (tp);
339             if (tp->t_outq.c_cc > SLIP_HIWAT)
340                 return;
341         }
342         /*
343          * This happens briefly when the line shuts down.
344          */
345         if (sc == NULL)
346             return;
347         /*
348          * Get a packet and send it to the interface.
349          */
350         s = splimp();
351         IF_DEQUEUE(&sc->sc_fastq, m);
352         if (m)
353             sc->sc_if.if_omcasts++; /* XXX */
354         else
355             IF_DEQUEUE(&sc->sc_if.if_snd, m);
356         splx(s);
357         if (m == NULL)
358             return;
359         /*
360          * We do the header compression here rather than in sloutput
361          * because the packets will be out of order if we are using TOS
362          * queueing, and the connection id compression will get
363          * munged when this happens.
364          */
365         if (sc->sc_bpf) {
366             /*
367              * We need to save the TCP/IP header before it's
368              * compressed. To avoid complicated code, we just
369              * 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 *m1 = m;
375         u_char *cp = bpfbuf + SLIP_HDRLEN;
376
377         len = 0;
378         do {
379             int mlen = m1->m_len;
380             bcopy(mtod(m1, caddr_t), cp, mlen);
381             cp += mlen;
382             len += mlen;
383             } while (m1 = m1->m_next);
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         }
399
400         /* packet output code */
401
402     }
403 }
404 }

```

if\_sl.c

Figure 5.16 slstart function: packet dequeuing.

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.

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.

```

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 (cfreecount < 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*

*if\_sl.c*

Figure 5.17 slstart function: resource shortages and line noise.

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.

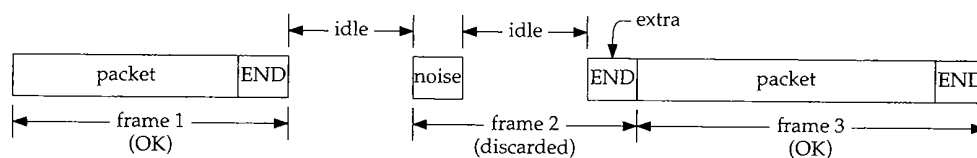


Figure 5.18 Karn's method for discarding noise on a SLIP line.

In Figure 5.19 there is no noise on the line and the 0-length frame is discarded by the receiving system.

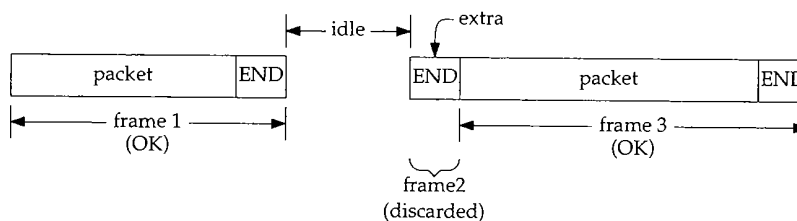


Figure 5.19 Karn's method with no noise.

The next section of slstart (Figure 5.20) transfers the data from an mbuf to the output queue for the TTY device.

```

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 (putc(*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*

Figure 5.20 slstart function: packet transmission.

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.

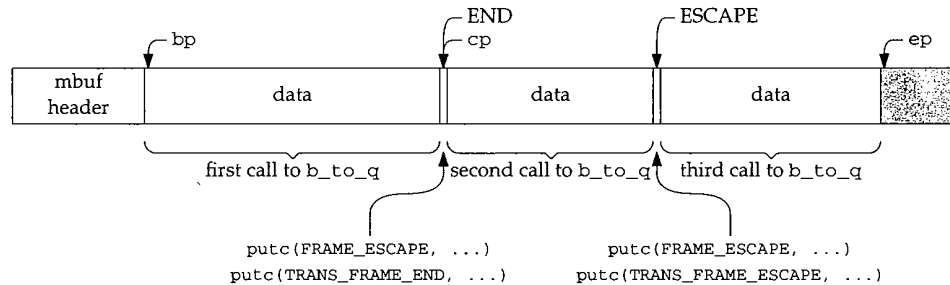


Figure 5.21 SLIP transmission of a single mbuf.

*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.

```

468     if (putc(FRAME_END, &tp->t_outq)) {                                     if_sl.c
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     }

```

Figure 5.22 *slstart* function: end-of-frame processing.

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 `s1start` 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.

2. 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.

3. 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.

```

210 void
211 slclose(tp)
212 struct tty *tp;
213 {
214     struct sl_softc *sc;
215     int s;
216
217     ttywflush(tp);
218     s = splimp(); /* actually, max(spltty, splnet) */
219     tp->t_line = 0;
220     sc = (struct sl_softc *) tp->t_sc;
221     if (sc != NULL) {
222         if_down(&sc->sc_if);
223         sc->sc_ttyp = NULL;
224         tp->t_sc = NULL;
225         MCLFREE((caddr_t) (sc->sc_ep - SLBUFSIZE));
226         sc->sc_ep = 0;
227         sc->sc_mp = 0;
228         sc->sc_buf = 0;
229     }
230     splx(s);
231 }

```

Figure 5.23 `slclose` function.

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.

**sltioctl 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 `sliocntl` processes `ioctl` commands issued for a SLIP interface through a socket descriptor. In Section 4.4 we showed how `ifioctl` calls `sliocntl`. We'll see a similar pattern for `ioctl` commands that we cover in later chapters.

Figure 5.7 also indicated that `sltioctl` processes `ioctl` commands issued for the TTY device associated with a SLIP network interface. The one command recognized by `sltioctl` is shown in Figure 5.24.

Command	Argument	Function	Description
SLIOCGUNIT	int *	sltioctl	return interface unit associated with the TTY device

Figure 5.24 `sltioctl` commands.

The `sltioctl` function is shown in Figure 5.25.

```

236 int
237 sltioctl(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*

*if\_sl.c*

Figure 5.25 `sltioctl` function.

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.

## 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.

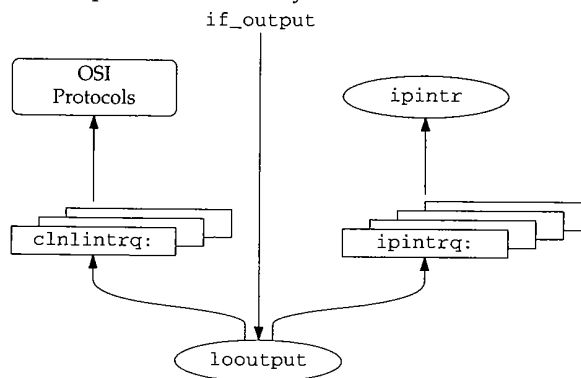


Figure 5.26 Loopback device driver.

`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 also be looped back in this way. `looutput` is shown in Figure 5.27.

```

57 int
58 looutput(ifp, m, dst, rt)
59 struct ifnet *ifp;
60 struct mbuf *m;
61 struct sockaddr *dst;
62 struct rtable *rt;
63 {
64     int    s, isr;
65     struct ifqueue *ifq = 0;
66
67     if ((m->m_flags & M_PKTHDR) == 0)
68         panic("looutput no HDR");
69     ifp->if_lastchange = time;
70     if (loif.if_bpf) {
71         /*
72          * We need to prepend the address family as
73          * 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     m0.m_next = m;
80     m0.m_len = 4;
81     m0.m_data = (char *) &af;

82     bpf_mtap(loif.if_bpf, &m0);
83 }
84 m->m_pkthdr.rcvif = ifp;

85 if (rt && rt->rt_flags & (RTF_REJECT | RTF_BLACKHOLE)) {
86     m_freem(m);
87     return (rt->rt_flags & RTF_BLACKHOLE ? 0 :
88           rt->rt_flags & RTF_HOST ? EHOSTUNREACH : ENETUNREACH);
89 }
90 ifp->if_opackets++;
91 ifp->if_obytes += m->m_pkthdr.len;
92 switch (dst->sa_family) {
93 case AF_INET:
94     ifq = &ipintrq;
95     isr = NETISR_IP;
96     break;

97 case AF_ISO:
98     ifq = &clnlintrq;
99     isr = NETISR_ISO;
100    break;

101 default:
102     printf("lo%d: can't handle af%d\n", ifp->if_unit,
103           dst->sa_family);
104     m_freem(m);
105     return (EAFNOSUPPORT);
106 }
107 s = splimp();
108 if (IF_QFULL(ifq)) {
109     IF_DROP(ifq);
110     m_freem(m);
111     splx(s);
112     return (ENOBUFS);
113 }
114 IF_ENQUEUE(ifq, m);
115 schednetisr(isr);
116 ifp->if_ipackets++;
117 ifp->if_ibytes += m->m_pkthdr.len;
118 splx(s);
119 return (0);
120 }

```

*if\_loop.c*

Figure 5.27 The looutput function.

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.

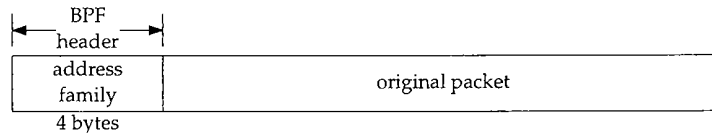


Figure 5.28 BPF loopback packet: logical format.

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.

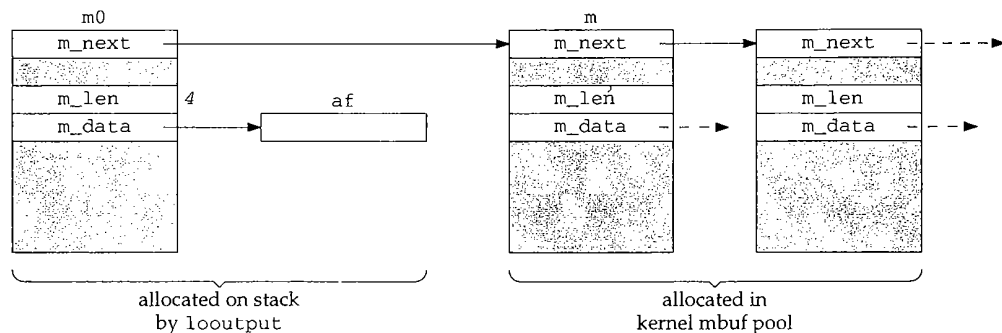


Figure 5.29 BPF loopback packet: mbuf format.

`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 `schednet_isr`.

## 5.5 Summary

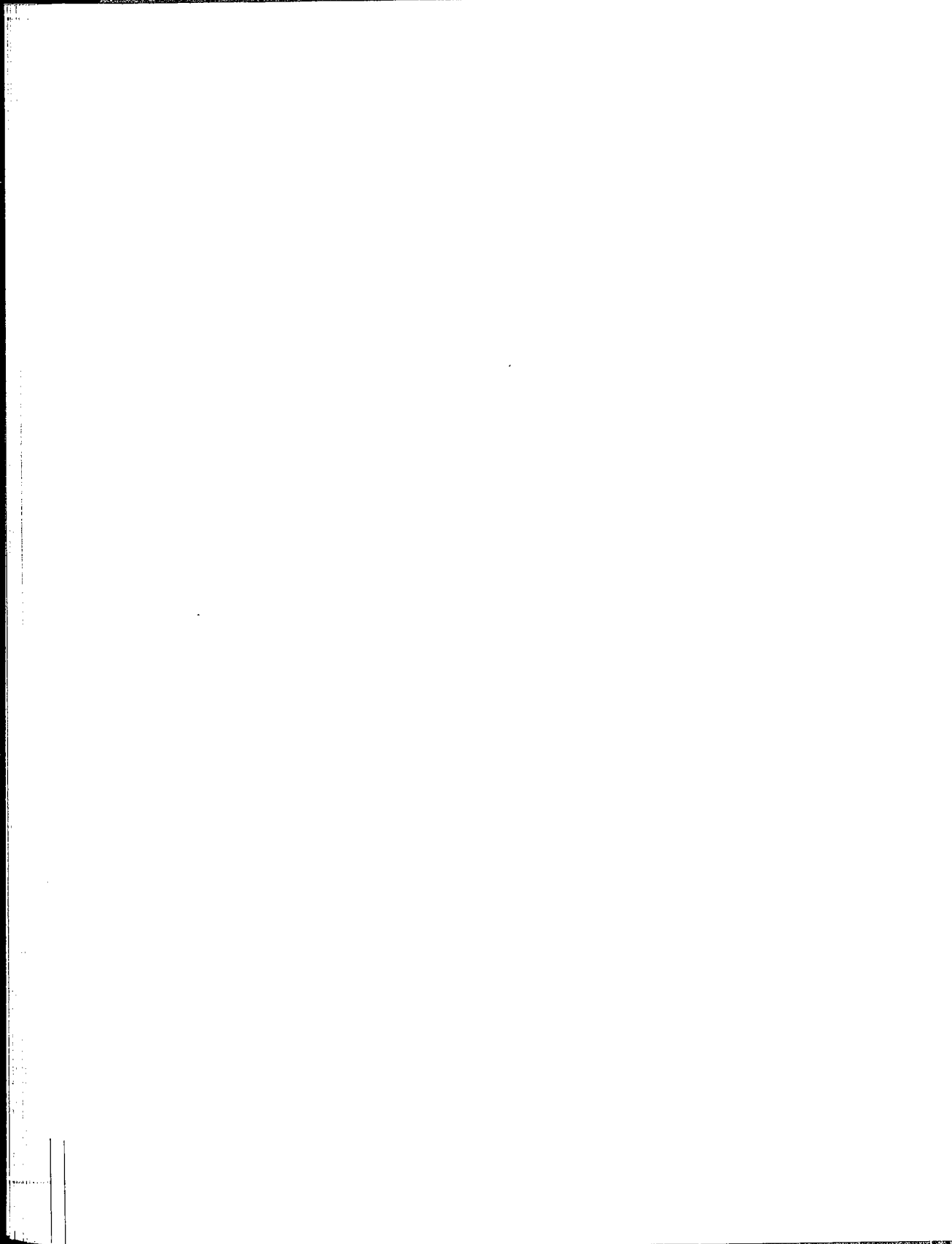
We described the two remaining interfaces to which we refer throughout the text: `sl0`, a SLIP interface, and `lo0`, the standard loopback interface.

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?
- 5.6 An earlier version of `slinput` did not set `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?



# 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.

Class	Range	Type
A	0.0.0.0 to 127.255.255.255	unicast
B	128.0.0.0 to 191.255.255.255	
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

Figure 6.1 Ranges for different classes of IP addresses.

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).

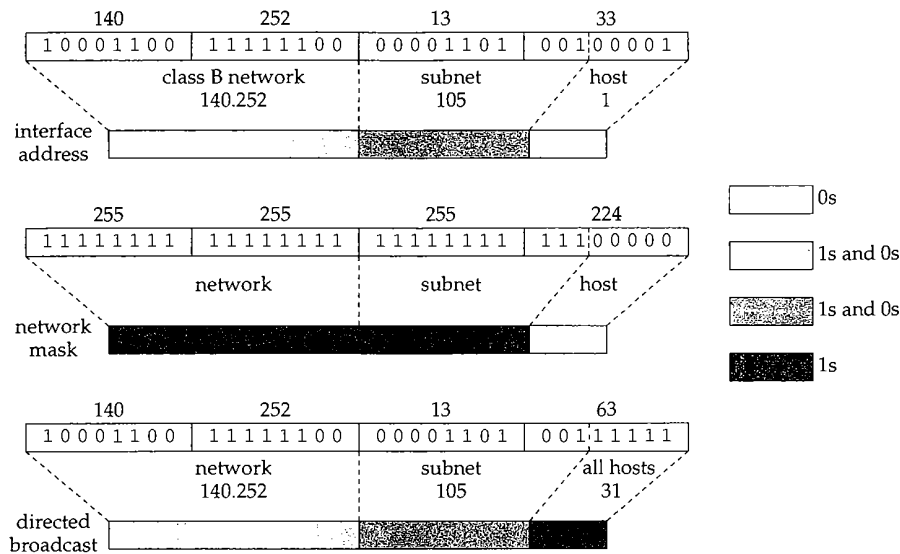


Figure 6.2 Alternate IP address notations.

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.

## 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.

File	Description
<code>netinet/in.h</code>	Internet address definitions
<code>netinet/in_var.h</code>	Internet interface definitions
<code>netinet/in.c</code>	Internet initialization and utility functions
<code>netinet/if.c</code>	Internet interface utility functions

Figure 6.3 Files discussed in this chapter.

### Global Variables

The two global variables introduced in this chapter are listed in Figure 6.4.

Variable	Datatype	Description
<code>in_ifaddr</code>	<code>struct in_ifaddr *</code>	head of <code>in_ifaddr</code> structure list
<code>in_interfaces</code>	<code>int</code>	number of IP capable interfaces

Figure 6.4 Global variables introduced in this chapter.

## 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 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.



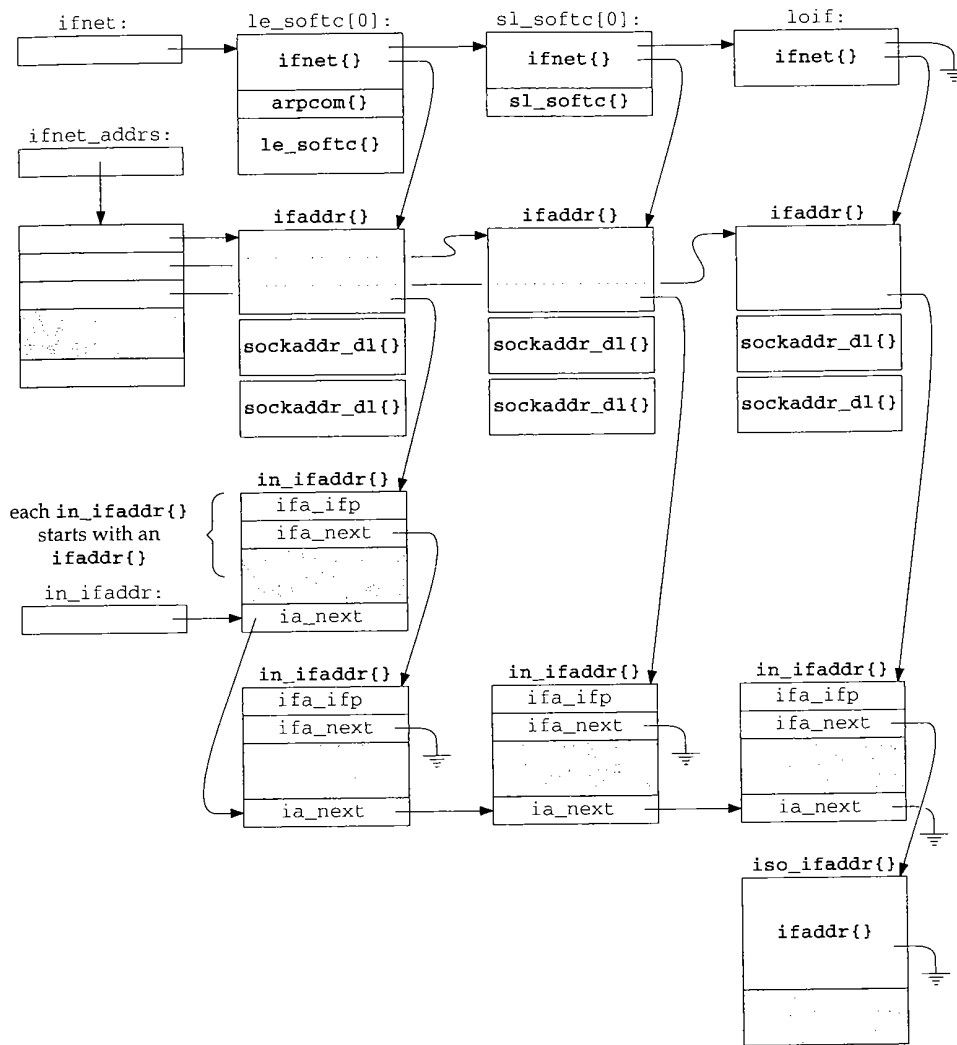


Figure 6.5 Interface and address data structures.

The following sections describe the data structures contained in Figure 6.5 and the IP-specific `ioctl` commands that examine and modify the structures.

## 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:

```

68 struct in_addr {
69     u_long  s_addr;           /* 32-bit IP address, net byte order */
70 };
                                     -----in.h

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

```

Figure 6.6 `sockaddr_in` structure.

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.

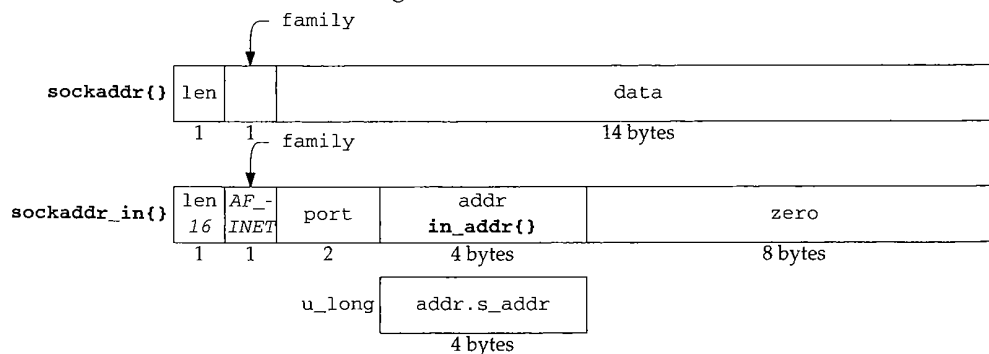


Figure 6.7 The organization of a `sockaddr_in` structure (`sin_omitted`).

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.

## 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).

```

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*

---

*in\_var.h*

Figure 6.8 The `in_ifaddr` structure.

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.

55-56 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.

## 6.6 Address Assignment

In Chapter 4 we showed the initialization of the interface structures when they are recognized at system initialization time. Before the Internet protocols can communicate through the interfaces, they must be assigned an IP address. Once the Net/3 kernel is



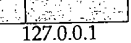
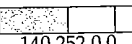

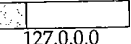


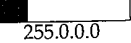
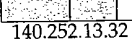
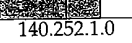
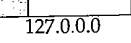

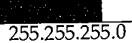
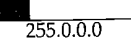



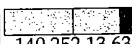
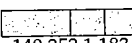
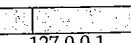


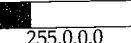
Variable	Type	Ethernet	PPP	Loopback	Description
ia_addr	sockaddr_in	 140.252.13.33	 140.252.1.29	 127.0.0.1	network, subnet, and host numbers
ia_net	u_long	 140.252.0.0	 140.252.0.0	 127.0.0.0	network number
ia_netmask	u_long	 255.255.0.0	 255.255.0.0	 255.0.0.0	network number mask
ia_subnet	u_long	 140.252.13.32	 140.252.1.0	 127.0.0.0	network and subnet number
ia_subnetmask	u_long	 255.255.255.224	 255.255.255.0	 255.0.0.0	network and subnet mask
ia_netbroadcast	in_addr	 140.252.255.255	 140.252.255.255	 127.255.255.255	network broadcast address
ia_broadaddr	sockaddr_in	 140.252.13.63			directed broadcast address
ia_dstaddr	sockaddr_in		 140.252.1.183	 127.0.0.1	destination address
ia_sockmask	sockaddr_in	 255.255.255.224	 255.255.255.0	 255.0.0.0	like ia_subnetmask but in network byte order

Figure 6.9 Ethernet, PPP, and loopback in\_ifaddr structures on sun.

running, the interfaces are configured by the `ifconfig` program, which issues configuration commands through the `ioctl` system call on a socket. This is normally done by the `/etc/netstart` shell script, which is executed when the system is bootstrapped.

Figure 6.10 shows the `ioctl` commands discussed in this chapter. The addresses associated with the commands must be from the same address family supported by the socket on which the commands are issued (i.e., you can't configure an OSI address through a UDP socket). For IP addresses, the `ioctl` commands are issued on a UDP socket.

Command	Argument	Function	Description
<code>SIOCGIFADDR</code>	struct ifreq *	in_control	get interface address
<code>SIOCGIFNETMASK</code>	struct ifreq *	in_control	get interface netmask
<code>SIOCGIFDSTADDR</code>	struct ifreq *	in_control	get interface destination address
<code>SIOCGIFBRDADDR</code>	struct ifreq *	in_control	get interface broadcast address
<code>SIOCSIFADDR</code>	struct ifreq *	in_control	set interface address
<code>SIOCSIFNETMASK</code>	struct ifreq *	in_control	set interface netmask
<code>SIOCSIFDSTADDR</code>	struct ifreq *	in_control	set interface destination address
<code>SIOCSIFBRDADDR</code>	struct ifreq *	in_control	set interface broadcast address
<code>SIOCDELIFADDR</code>	struct ifreq *	in_control	delete interface address
<code>SIOCAIFADDR</code>	struct in_aliasreq *	in_control	add interface address

Figure 6.10 Interface `ioctl` commands.

The commands that get address information start with SIOCG, and the commands that set address information start with SIOCS. SIOC stands for *socket ioctl*, the G for *get*, and the S for *set*.

In Chapter 4 we looked at five *protocol-independent ioctl* commands. The commands in Figure 6.10 modify the addressing information associated with an interface. Since addresses are protocol-specific, the command processing is *protocol-dependent*. Figure 6.11 highlights the *ioctl*-related functions associated with these commands.

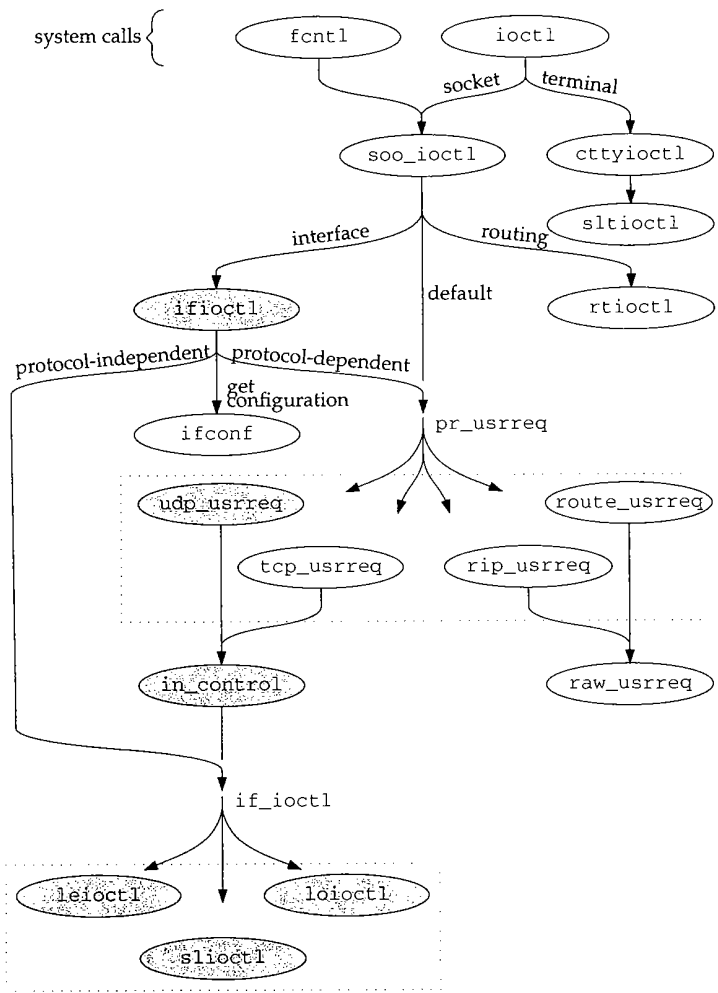


Figure 6.11 *ioctl* functions described in this chapter.

### ifioctl Function

As shown in Figure 6.11, `ifioctl` passes protocol-dependent `ioctl` commands to the `pr_usrreq` function of the protocol associated with the socket. Control is passed to `udp_usrreq` and immediately to `in_control` where most of the processing occurs. If the same commands are issued on a TCP socket, control would also end up at `in_control`. Figure 6.12 repeats the default code from `ifioctl`, first shown in Figure 4.22.

```

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 }

```

Figure 6.12 `ifioctl` function: protocol-specific commands.

447-454 The function passes all the relevant data for the `ioctl` commands listed in Figure 6.10 to the user-request function of the protocol associated with the socket on which the request was made. For a UDP socket, `udp_usrreq` is called. Section 23.10 describes the `udp_usrreq` function in detail. For now, we need to look only at the `PRU_CONTROL` code from `udp_usrreq`:

```

if (req == PRU_CONTROL)
    return (in_control(so, (int)m, (caddr_t)addr, (struct ifnet *)control));

```

### in\_control Function

Figure 6.11 shows that control can reach `in_control` through the default case in `soo_ioctl` or through the protocol-dependent case in `ifioctl`. In both cases, `udp_usrreq` calls `in_control` and returns whatever `in_control` returns. Figure 6.13 shows `in_control`.

132-145 `so` points to the socket on which the `ioctl` (specified by the second argument, `cmd`) was issued. The third argument, `data`, points to the data (second column of Figure 6.10) to be used or returned by the command. The last argument, `ifp`, is null (non-interface `ioctl` from `soo_ioctl`) or points to the interface named in the `ifreq` or `in_aliasreq` structures (interface `ioctl` from `ifioctl`). `in_control` initializes `ifa` and `ifra` to access data as an `ifreq` or as an `in_aliasreq` structure.

146-152 If `ifp` points to an `ifnet` structure, the for loop locates the *first* address on the Internet address list associated with the interface. If an address is found, `ia` points to its `in_ifaddr` structure, otherwise, `ia` is null.

If `ifp` is null, `cmd` will not match any of the cases in the first switch or any of the nondefault cases in the second switch. The default case in the second switch returns `EOPNOTSUPP` when `ifp` is null.

```

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) {

        /* establish preconditions for commands */

218     }
219     switch (cmd) {

        /* perform the commands */

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 }

```

Figure 6.13 in\_control function.

153-330 The first switch in `in_control` makes sure all the preconditions for each command are met before the second switch processes the command. The individual cases are described in the following sections.

If the default case is executed in the second switch, `ifp` points to an interface structure, and the interface has an `if_ioctl` function, then `in_control` passes the `ioctl` command to the interface for device-specific processing.

Net/3 does not define any interface commands that would be processed by the default case. But the driver for a particular device might define its own interface `ioctl` commands and they would be processed by this case.

331-332 We'll see that many of the cases within the `switch` statements return directly. If control falls through both `switch` statements, `in_control` returns 0. Several of the cases do break out of the second `switch`.

We look at the interface `ioctl` commands in the following order:

- assigning an address, network mask, or destination address;
- assigning a broadcast address;
- retrieving an address, network mask, destination address, or broadcast address;
- assigning multiple addresses to an interface; or
- deleting an address.

For each group of commands, we describe the precondition processing done in the first `switch` statement and then the command processing done in the second `switch` statement.

#### Preconditions: `SIOCSIFADDR`, `SIOCSIFNETMASK`, and `SIOCSIFDSTADDR`

Figure 6.14 shows the precondition testing for `SIOCSIFADDR`, `SIOCSIFNETMASK`, and `SIOCSIFDSTADDR`.

##### Superuser only

166-172 If the socket was not created by a superuser process, these commands are prohibited and `in_control` returns `EPERM`. If no interface is associated with the request, the kernel panics. The panic should never happen since `ifioctl` returns if it can't locate an interface (Figure 4.22).

The `SS_PRIV` flag is set by `socreate` (Figure 15.16) when a superuser process creates a socket. Because the test here is against the flag and not the effective user ID of the process, a set-user-ID root process can create a socket, and give up its superuser privileges, but still issue privileged `ioctl` commands.

##### Allocate structure

173-191 If `ia` is null, the command is requesting a new address. `in_control` allocates an `in_ifaddr` structure, clears it with `bzero`, and links it into the `in_ifaddr` list for the system and into the `if_addrlist` list for the interface.

##### Initialize structure

192-201 The next portion of code initializes the `in_ifaddr` structure. First the generic pointers in the `ifaddr` portion of the structure are initialized to point to the `sockaddr_in` structures in the `in_ifaddr` structure. The function also initializes the `ia_sockmask` and `ia_broadaddr` structures as necessary. Figure 6.15 illustrates the `in_ifaddr` structure after this initialization.

202-206 Finally, `in_control` establishes the back pointer from the `in_ifaddr` to the interface's `ifnet` structure.

Net/3 counts only nonloopback interfaces in `in_interfaces`.



```

166     case SIOCSIFADDR:
167     case SIOCSIFNETMASK:
168     case SIOCSIFDSTADDR:
169         if ((so->so_state & SS_PRIV) == 0)
170             return (EPERM);

171         if (ifp == 0)
172             panic("in_control");
173         if (ia == (struct in_ifaddr *) 0) {
174             oia = (struct in_ifaddr *)
175                 malloc(sizeof *oia, M_IFADDR, M_WAITOK);
176             if (oia == (struct in_ifaddr *) NULL)
177                 return (ENOBUFS);
178             bzero((caddr_t) oia, sizeof *oia);
179             if (ia = in_ifaddr) {
180                 for (; ia->ia_next; ia = ia->ia_next)
181                     continue;
182                 ia->ia_next = oia;
183             } else
184                 in_ifaddr = oia;
185             ia = oia;
186             if (ifa = ifp->if_addrlist) {
187                 for (; ifa->ifa_next; ifa = ifa->ifa_next)
188                     continue;
189                 ifa->ifa_next = (struct ifaddr *) ia;
190             } else
191                 ifp->if_addrlist = (struct ifaddr *) ia;

192             ia->ia_ifa.ifa_addr = (struct sockaddr *) &ia->ia_addr;
193             ia->ia_ifa.ifa_dstaddr
194                 = (struct sockaddr *) &ia->ia_dstaddr;
195             ia->ia_ifa.ifa_netmask
196                 = (struct sockaddr *) &ia->ia_sockmask;
197             ia->ia_sockmask.sin_len = 8;
198             if (ifp->if_flags & IFF_BROADCAST) {
199                 ia->ia_broadaddr.sin_len = sizeof(ia->ia_addr);
200                 ia->ia_broadaddr.sin_family = AF_INET;
201             }
202             ia->ia_ifp = ifp;
203             if (ifp != &loif)
204                 in_interfaces++;
205         }
206         break;

```

Figure 6.14 in\_control function: address assignment.

### Address Assignment: SIOCSIFADDR

The precondition code has ensured that *ia* points to an *in\_ifaddr* structure to be modified by the SIOCSIFADDR command. Figure 6.16 shows the code executed by *in\_control* in the second switch for this command.

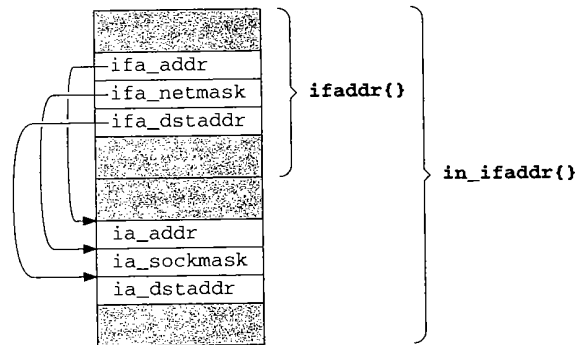


Figure 6.15 An `in_ifaddr` structure after initialization by `in_control`.

```

259     case SIOCSIFADDR:
260         return (in_ifinit(ifp, ia,
261             (struct sockaddr_in *) &ifr->ifr_addr, 1));

```

Figure 6.16 `in_control` function: address assignment.

259-261 `in_ifinit` does all the work. The IP address included within the `ifreq` structure (`ifr_addr`) is passed to `in_ifinit`.

### `in_ifinit` Function

The major steps in `in_ifinit` are:

- copy the address into the structure and inform the hardware of the change,
- discard any routes configured with the previous address,
- establish a subnet mask for the address,
- establish a default route to the attached network (or host), and
- join the all-hosts group on the interface.

The code is described in three parts, starting with Figure 6.17.

353-359 The four arguments to `in_ifinit` are: `ifp`, a pointer to the interface structure; `ia`, a pointer to the `in_ifaddr` structure to be changed; `sin`, a pointer to the requested IP address; and `scrub`, which indicates if existing routes for this interface should be discarded. `i` holds the IP address in host byte order.

#### Assign address and notify hardware

360-374 `in_ifinit` saves the previous address in `oldaddr` in case it must be restored when an error occurs. If the interface has an `if_ioctl` function defined, `in_control` calls it. The three functions `leioctl`, `slioclt`, and `loioctl` for the sample interfaces are described in the next section. The previous address is restored and `in_control` returns if an error occurs.

```

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 |= RTF_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     }

```

Figure 6.17 in\_ifinit function: address assignment and route initialization.

### Ethernet configuration

375-378 For Ethernet devices, `arp_rtrequest` is selected as the link-level routing function and the `RTF_CLONING` flag is set. `arp_rtrequest` is described in Section 21.13 and `RTF_CLONING` is described at the end of Section 19.4. As the XXX comment suggests, putting the code here avoids changing all the Ethernet drivers.

### Discard previous routes

379-384 If the caller requests that existing routes be scrubbed, the previous address is reattached to `ifa_addr` while `in_ifscrub` locates and invalidates any routes based on the old address. After `in_ifscrub` returns, the new address is restored.

The section of `in_ifinit` shown in Figure 6.18 constructs the network and subnet masks.

```

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);

```

Figure 6.18 `in_ifinit` function: network and subnet masks.

#### Construct network mask and default subnetmask

385-400 A tentative network mask is constructed in `ia_netmask` based on whether the address is a class A, class B, or class C address. If no subnetwork mask is associated with the address yet, `ia_subnetmask` and `ia_sockmask` are initialized to the tentative mask in `ia_netmask`.

If a subnet has been specified, `in_ifinit` logically ANDs the tentative netmask and the existing submask together to get a new network mask. This operation may clear some of the 1 bits in the tentative netmask (it can never set the 0 bits, since 0 logically ANDed with anything is 0). In this case, the network mask has fewer 1 bits than would be expected by considering the class of the address.

This is called *supernetting* and is described in RFC 1519 [Fuller et al. 1993]. A supernet is a grouping of several class A, class B, or class C networks. Supernetting is also discussed in Section 10.8 of Volume 1.

An interface is configured by default *without subnetting* (i.e., the network and subnetwork masks are the same). An explicit request (with `SIOCSIFNETMASK` or `SIOCAIFADDR`) is required to enable subnetting (or supernetting).

#### Construct network and subnetwork numbers

401-403 The network and subnetwork numbers are extracted from the new address by the network and subnet masks. The function `in_socktrim` sets the length of `in_sockmask` (which is a `sockaddr_in` structure) 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 a route for the interface and joins the all-hosts multicast group.

```

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  return (error);
434 }

```

Figure 6.19 `in_ifinit` function: routing and multicast groups.

#### Establish route for host or network

404-422 The next step is to create a route for the network specified by the new address. `in_ifinit` copies the routing metric from the interface to the `in_ifaddr` structure, constructs the broadcast addresses if the interface supports broadcasts, and forces the destination address to be the same as the assigned address for loopback interfaces. If a point-to-point interface does not yet have an IP address assigned to the other end of the link, `in_ifinit` returns before trying to establish a route for the invalid address.

`in_ifinit` initializes `flags` to `RTF_UP` and logically ORs in `RTF_HOST` for loopback and point-to-point interfaces. `rtinit` installs a route to the network (`RTF_HOST` not set) or host (`RTF_HOST` set) for the interface. If `rtinit` succeeds, the `IFA_ROUTE` flag in `ia_flags` is set to indicate that a route is installed for this address.

#### Join all-hosts group

423-433 Finally, a multicast capable interface must join the all-hosts multicast group when it is initialized. `in_addmulti` does the work and is described in Section 12.11.

**Network Mask Assignment: SIOCSIFNETMASK**

Figure 6.20 shows the processing for the network mask command.

```

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;

```

Figure 6.20 `in_control` function: network mask assignment.

262-265 `in_control` extracts the requested netmask from the `ifreq` structure and stores it in `ia_sockmask` in network byte order and in `ia_subnetmask` in host byte order.

**Destination Address Assignment: SIOCSIFDSTADDR**

For point-to-point interfaces, the address of the system on the other end of the link is specified by the `SIOCSIFDSTADDR` command. Figure 6.14 showed the precondition processing for the code shown in Figure 6.21.

```

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;

```

Figure 6.21 `in_control` function: destination address assignment.

236-245 Only point-to-point networks have destination addresses, so `in_control` returns `EINVAL` for other networks. After saving the current destination address in `oldaddr`, the code sets the new address and informs the interface through the `if_ioctl` function. If an error occurs, the old address is restored.

246-253 If the address has a route previously associated with it, that route is deleted by the first call to `rtinit` and a new route to the new destination is installed by the second call to `rtinit`.

## Retrieving Interface Information

Figure 6.22 shows the precondition processing for the SIOCSIFBRDADDR command as well as the ioctl commands that return interface information to the calling process.

```

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

Figure 6.22 in\_control function: preconditions.

207-217 The broadcast address may only be set through a socket created by a superuser process. The SIOCSIFBRDADDR command and the four SIOCGxxx commands work only when an address is already defined for the interface, in which case ia won't be null (ia was set by in\_control, Figure 6.13). If ia is null, EADDRNOTAVAIL is returned.

The processing of these five commands (four get commands and one set command) is shown in Figure 6.23.

```

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_dstaddr) = 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

Figure 6.23 in\_control function: processing.

220-235 The unicast address, broadcast address, destination address, or netmask are copied into the `ifreq` structure. A broadcast address is available only from a network interface that supports broadcasts, and a destination address is available only from a point-to-point interface.

254-258 The broadcast address is copied from the `ifreq` structure only when the interface supports broadcasts.

### Multiple IP Addresses per Interface

The `SIOCGxxx` and `SIOCSxxx` commands operate only on the first IP address associated with an interface—the first address located by the loop at the start of `in_control` (Figure 6.25). To support multiple IP addresses per interface, the additional addresses must be assigned and configured with the `SIOCAIFADDR` command. In fact, `SIOCAIFADDR` can do everything the `SIOCGxxx` and `SIOCSxxx` commands do. The `ifconfig` program uses `SIOCAIFADDR` to configure all of the address information for an interface.

As noted earlier, having multiple addresses per interface can ease the transition when hosts or networks are renumbered. A fault-tolerant software system might use this feature to allow a backup system to assume the IP address of a failed system.

The `-alias` option to Net/3's `ifconfig` program passes information about the additional addresses to the kernel in an `in_aliasreq` structure, shown in 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 };

```

in\_var.h

in\_var.h

Figure 6.24 in\_aliasreq structure.

59-65 Notice that unlike the `ifreq` structure, there is no union defined within the `in_aliasreq` structure. With `SIOCAIFADDR`, the address, broadcast address, and mask can be specified in a single `ioctl` call.

`SIOCAIFADDR` adds a new address or changes the information associated with an existing address. `SIOCDEFADDR` deletes the `in_ifaddr` structure for the matching IP address. Figure 6.25 shows the precondition processing for the `SIOCAIFADDR` and `SIOCDEFADDR` commands, which assumes that the loop at the start of `in_control` (Figure 6.13) has set `ia` to point to the *first* IP address associated with the interface specified in `ifra_name` (if it exists).



```

154     case SIOCAIFADDR:
155     case SIOCDEFADDR:
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 == SIOCDEFADDR && ia == 0)
164             return (EADDRNOTAVAIL);
165         /* FALLTHROUGH to Figure 6.14 */

```

Figure 6.25 `in_control` function: adding and deleting addresses.

154-165 Because the `SIOCDEFADDR` code looks only at the first two members of `*ifra`, the code shown in Figure 6.25 works for `SIOCAIFADDR` (when `ifra` points to an `in_aliasreq` structure) and for `SIOCDEFADDR` (when `ifra` points to an `ifreq` structure). The first two members of the `in_aliasreq` and `ifreq` structures are identical.

For both commands, the `for` loop continues the search started by the loop at the start of `in_control` by looking for the `in_ifaddr` structure with the same IP address specified by `ifra->ifra_addr`. For the delete command, `EADDRNOTAVAIL` is returned if the address isn't found.

After the loop and the test for the delete command, control falls through to the code we described in Figure 6.14. For the add command, the code in Figure 6.14 allocates a new `in_ifaddr` structure if one was not found that matched the address in the `in_aliasreq` structure.

#### Additional IP Addresses: `SIOCAIFADDR`

At this point `ia` points to a new `in_ifaddr` structure or to an old `in_ifaddr` structure with an IP address that matched the address in the request. The `SIOCAIFADDR` processing is shown in Figure 6.26.

266-277 Since `SIOCAIFADDR` can create a new address or change the information associated with an existing address, the `maskIsNew` and `hostIsNew` flags keep track of what has changed so that routes can be updated if necessary at the end of the function.

By default, the code assumes that a new IP address is being assigned to the interface (`hostIsNew` starts at 1). If the length of the new address is 0, `in_control` copies the current address into the request and changes `hostIsNew` to 0. If the length is not 0 and the new address matches the old address, this request does not contain a new address and `hostIsNew` is set to 0.

278-284 If a netmask is specified in the request, any routes using the current address are discarded and `in_control` installs the new mask.

285-290 If the interface is a point-to-point interface and the request includes a new destination address, `in_scrub` discards any routes using the address, the new destination address is installed, and `maskIsNew` is set to 1 to force the call to `in_ifinit`, which reconfigures the interface.

```

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);

```

Figure 6.26 in\_control function: SIOCAIFADDR processing.

291-297 If a new address has been configured or a new mask has been assigned, `in_ifinit` makes all the appropriate changes to support the new configuration (Figure 6.17). Note that the last argument to `in_ifinit` is 0. This indicates that it isn't necessary to scrub any routes since that has already been taken care of. Finally, the broadcast address is copied from the `in_aliasreq` structure if the interface supports broadcasts.

### Deleting IP Addresses: SIOCDEFADDR

The `SIOCDEFADDR` command, which deletes IP addresses from an interface, is shown in Figure 6.27. Remember that `ia` points to the `in_ifaddr` structure to be deleted (i.e., the one that matched the request).

298-323 The precondition code arranged for `ia` to point to the address to be deleted. `in_ifscrub` deletes any routes associated with the address. The first `if` deletes the

```

298     case SIOCIFADDR:
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;

```

Figure 6.27 in\_control function: deleting addresses.

structure for the interface address list. The second `if` deletes the structure from the Internet address list (`in_ifaddr`).

324-325 IFAFREE only releases the structure when the reference count drops to 0.

The additional references would be from entries in the routing table.

## 6.7 Interface ioctl Processing

We now look at the specific `ioctl` processing done by each of our sample interfaces in the `leiioctl`, `sliioctl`, and `loiioctl` 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.

```

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 lereg1 *ler1 = le->sc_r1;
622     int s = splimp(), error = 0;
623
624     switch (cmd) {
625     case SIOCSIFADDR:
626         ifp->if_flags |= IFF_UP;
627         switch (ifa->ifa_addr->sa_family) {
628         case AF_INET:
629             leinit(ifp->if_unit); /* before arpwhohas */
630             ((struct arpcom *) ifp)->ac_ipaddr =
631                 IA_SIN(ifa)->sin_addr;
632             arpwhohas((struct arpcom *) ifp, &IA_SIN(ifa)->sin_addr);
633             break;
634         default:
635             leinit(ifp->if_unit);
636             break;
637         }
638     }
639
640     /* SIOCSIFFLAGS command (Figure 4.31) */
641     /* SIOCADMULTI and SIOCDELMULTI commands (Figure 12.31) */
642
643     default:
644         error = EINVAL;
645     }
646     splx(s);
647     return (error);
648 }

```

if\_le.c

Figure 6.28 leioctl function.

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.

**slioct1 Function**

The `slioct1` function (Figure 6.29) processes the `SIOCSIFADDR` and `SIOCSIFDSTADDR` command for the SLIP device driver.

```

653 int
654 slioct1(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
663     switch (cmd) {
664     case SIOCSIFADDR:
665         if (ifa->ifa_addr->sa_family == AF_INET)
666             ifp->if_flags |= IFF_UP;
667         else
668             error = EAFNOSUPPORT;
669         break;
670
671     case SIOCSIFDSTADDR:
672         if (ifa->ifa_addr->sa_family != AF_INET)
673             error = EAFNOSUPPORT;
674         break;
675
676     /* SIOCADDMULTI and SIOCDELMULTI commands (Figure 12.29) */
677
678     default:
679         error = EINVAL;
680     }
681     splx(s);
682     return (error);
683 }

```

**Figure 6.29** `slioct1` function: `SIOCSIFADDR` and `SIOCSIFDSTADDR` commands.

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.

**loioctl Function**

The `loioctl` function and its implementation of the `SIOCSIFADDR` command is shown in Figure 6.30.

```

135 int
136 loioctl(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         /* SIOCADMULTI and SIOCDELMULTI commands (Figure 12.30) */
155
156     default:
157         error = EINVAL;
158     }
159     return (error);
160 }

```

*if\_loop.c*

Figure 6.30 `loioctl` function: `SIOCSIFADDR` command.

135-151 For Internet addresses, `loioctl` 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`).

## 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`.

Function	Description
<code>in_netof</code>	Returns network and subnet portions of <code>in</code> . 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 <code>in</code> 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 <code>in</code> 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 <code>in</code> is a broadcast address associated with the interface pointed to by <code>ifp</code> .  <code>int in_broadcast(struct in_addr in, struct ifnet *ifp);</code>

Figure 6.31 Internet address functions.

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)

## 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`.

Function	Description
<code>ifa_ifwithaddr</code>	Search the <code>ifnet</code> list for an interface with a unicast or broadcast address of <code>addr</code> . Return a pointer to the matching <code>ifaddr</code> structure or a null pointer if no match is found.  <code>struct ifaddr * ifa_ifwithaddr(struct sockaddr *addr);</code>
<code>ifa_ifwithdstaddr</code>	Search the <code>ifnet</code> list for the interface with a destination address of <code>addr</code> . Return a pointer to the matching <code>ifaddr</code> structure or a null pointer if no match is found.  <code>struct ifaddr * ifa_ifwithdstaddr(struct sockaddr *addr);</code>
<code>ifa_ifwithnet</code>	Search the <code>ifnet</code> list for the address on the same network as <code>addr</code> . Return a pointer to the most specific matching <code>ifaddr</code> structure or a null pointer if no match is found.  <code>struct ifaddr * ifa_ifwithnet(struct sockaddr *addr);</code>
<code>ifa_ifwithaf</code>	Search the <code>ifnet</code> list for the first address in the same address family as <code>addr</code> . Return a pointer to the matching <code>ifaddr</code> structure or a null pointer if no match is found.  <code>struct ifaddr * ifa_ifwithaf(struct sockaddr *addr);</code>
<code>ifaof_ifpforaddr</code>	Search the address list of <code>ifp</code> for the address that matches <code>addr</code> . 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 <code>ifaddr</code> structure or a null pointer if no match is found.  <code>struct ifaddr * ifaof_ifpforaddr(struct sockaddr *addr, struct ifnet *ifp);</code>
<code>ifa_ifwithroute</code>	Returns a pointer to the <code>ifaddr</code> structure for the appropriate local interface for the destination ( <code>dst</code> ), and gateway ( <code>gateway</code> ) specified.  <code>struct ifaddr * ifa_ifwithroute(int flags, struct sockaddr *dst, struct sockaddr *gateway)</code>
<code>ifunit</code>	Return a pointer to the <code>ifnet</code> structure associated with <code>name</code> .  <code>struct ifnet * ifunit(char *name);</code>

Figure 6.32 ifnet utility functions.



## 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

- 6.1 Why do you think `sin_addr` in the `sockaddr_in` structure was originally defined as a structure?
- 6.2 `ifunit("s10")` returns a pointer to which structure in Figure 6.5?
- 6.3 Why is the IP address duplicated in `ac_ipaddr` when it is already contained in an `ifaddr` structure on the interface's address list?
- 6.4 Why do you think IP interface addresses are accessed through a UDP socket and not a raw IP socket?
- 6.5 Why does `in_socktrim` change `sin_len` to match the length of the mask instead of using the standard length of a `sockaddr_in` structure?
- 6.6 What happens when the connection request segment from a `telnet 127.0.0.2` command is erroneously forwarded by a Net/2 system and is eventually recognized and accepted by a system along the default route?

Table with multiple columns and rows, mostly blank or illegible. The table structure is visible in the bottom-left corner.

# 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.

Protocol family	Address family	Protocol
<i>PF_INET</i>	<i>AF_INET</i>	Internet
<i>PF_OSI, PF_ISO</i>	<i>AF_OSI, AF_ISO</i>	OSI
<i>PF_LOCAL, PF_UNIX</i>	<i>AF_LOCAL, AF_UNIX</i>	local IPC (Unix)
<i>PF_ROUTE</i>	<i>AF_ROUTE</i>	routing tables
n/a	<i>AF_LINK</i>	link-level (e.g., Ethernet)

Figure 7.1 Common protocol and address family constants.

*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.

## 7.2 Code Introduction

Two headers and two C files are covered in this chapter. Figure 7.2 describes the four files.

File	Description
netinet/domain.h	domain structure definition
netinet/protosw.h	protosw structure definition
netinet/in_proto.c	IP domain and protosw structures
kern/uipc_domain.c	initialization and search functions

Figure 7.2 Files discussed in this chapter.

### 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.

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

Figure 7.3 Global variables introduced in this chapter.

### 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.

Variable	Datatype	Description
ip_ifmatrix	int[][]	two-dimensional array to count packets routed between any two interfaces

Figure 7.4 Statistics collected in this chapter.

### 7.3 domain Structure

A protocol domain is represented by a domain structure shown in Figure 7.5.

```

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

Figure 7.5 The domain structure definition.

42-57 `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.

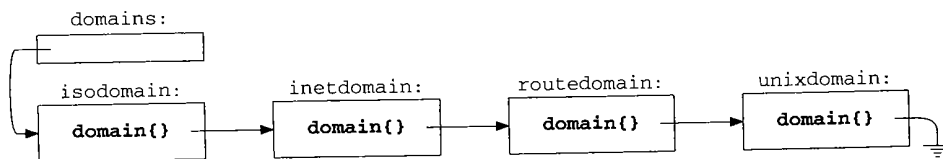


Figure 7.6 domains list.

## 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.

```

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*

*protosw.h*

Figure 7.7 The `protosw` structure definition.

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.

<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

Figure 7.8 `pr_type` specifies the protocol's semantics.

`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`.

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

Figure 7.9 pr\_flags values.

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.

pr_type	PR_			Record boundaries?	Reliable?	Example	
	ADDR	ATOMIC	CONNREQUIRED			Internet	Other
<i>SOCK_STREAM</i>			•	none	•	TCP	SPP
<i>SOCK_SEQPACKET</i>		•	•	explicit implicit	• •		TP4 SPP
<i>SOCK_RDM</i>		•	•	implicit	see text		RDP
<i>SOCK_DGRAM</i>	•	•		implicit		UDP	
<i>SOCK_RAW</i>	•	•		implicit		ICMP	

Figure 7.10 Protocol characteristics and examples.

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.

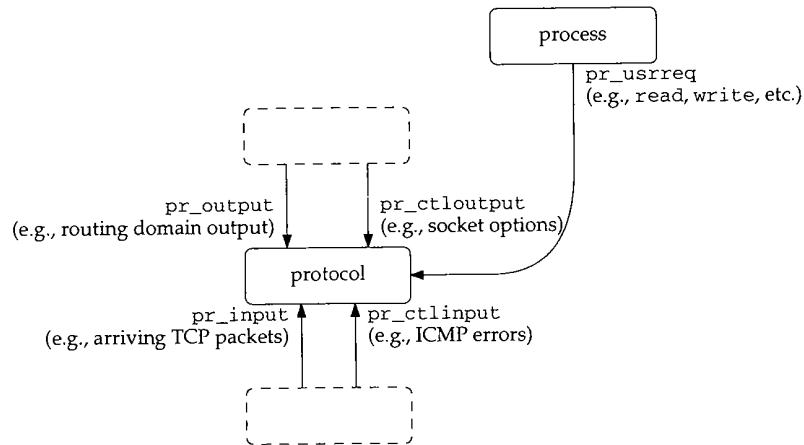


Figure 7.11 The five main entry points to a protocol.

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.

## 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.

<code>inetsw[]</code>	<code>pr_protocol</code>	<code>pr_type</code>	Description	Acronym
0	0	0	Internet Protocol	IP
1	<code>IPPROTO_UDP</code>	<code>SOCK_DGRAM</code>	User Datagram Protocol	UDP
2	<code>IPPROTO_TCP</code>	<code>SOCK_STREAM</code>	Transmission Control Protocol	TCP
3	<code>IPPROTO_RAW</code>	<code>SOCK_RAW</code>	Internet Protocol (raw)	IP (raw)
4	<code>IPPROTO_ICMP</code>	<code>SOCK_RAW</code>	Internet Control Message Protocol	ICMP
5	<code>IPPROTO_IGMP</code>	<code>SOCK_RAW</code>	Internet Group Management Protocol	IGMP
6	0	<code>SOCK_RAW</code>	Internet Protocol (raw, default)	IP (raw)

Figure 7.12 Internet domain protocols.

39-77 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.

```

39 struct protosw inetsw[] =
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_WANTRCVD,
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)};

```

Figure 7.13 The Internet domain and protosw structures.

78-81 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

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

Figure 7.14 The IP inetsw entries.

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.

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).

### 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.

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.

```

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_protoswNPROTOSW; 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 = MLEN - max_hdr;
66     timeout(pffasttimo, (void *) 0, 1);
67     timeout(pfslowtimo, (void *) 0, 1);
68 }

```

Figure 7.15 domaininit function.

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.

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.

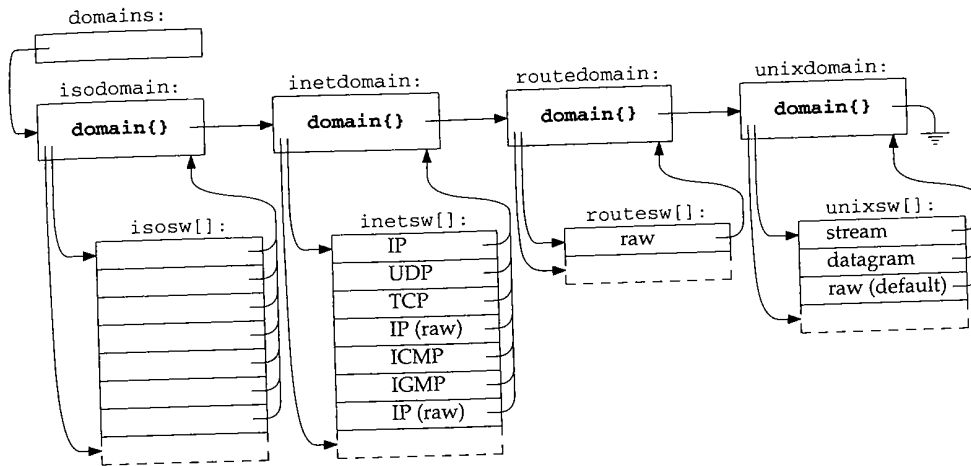


Figure 7.16 The domain list and protocols arrays after initialization.

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	max_linkhdr + max_protohdr
max_datalen	44	number of data bytes available in packet header mbuf after accounting for the link and protocol headers

Figure 7.17 Parameters used to minimize copying of protocol data.

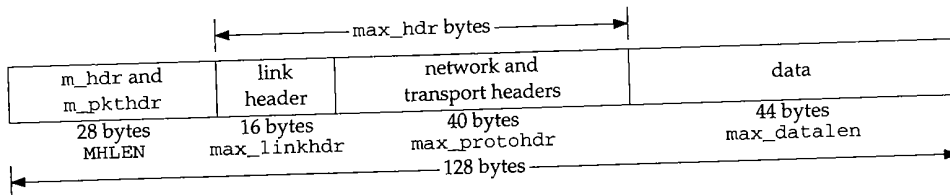


Figure 7.18 Mbuf and associated maximum header lengths.

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.

```

153 void
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 }

```

Figure 7.19 pfslowtimo and pffasttimo functions.

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.

## 7.6 pffindproto and pffindtype Functions

The `pffindproto` and `pffindtype` functions look up a protocol by number (e.g., `IPPROTO_TCP`) or by type (e.g., `SOCK_STREAM`). As we'll see in Chapter 15, these functions are called to locate the appropriate `protosw` entry when a process creates a socket.

69-84 `pffindtype` performs a linear search of domains for the specified family and then searches the protocols within the domain for the first one of the specified type.

85-107 `pffindproto` searches domains exactly as `pffindtype` does but looks for the family, type, and protocol specified by the caller. If `pffindproto` does not find a (protocol, type) match within the specified protocol family, and type is `SOCK_RAW`, and the domain has a default raw protocol (`pr_protocol` equals 0), then `pffindproto` selects the default raw protocol instead of failing completely. For example, a call such as

```

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 == 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 }

```

Figure 7.20 pffindproto and pffindtype functions.

```
pffindproto(PF_INET, 27, SOCK_RAW);
```

returns a pointer to `inetsw[6]`, the default raw IP protocol, since Net/3 does not include support for protocol 27. With access to raw IP, a process could implement protocol 27 services on its own using the kernel to manage the sending and receiving of the IP packets.

Protocol 27 is reserved for the Reliable Datagram Protocol (RFC 1151).

Both functions return a pointer to the `protosw` structure for the selected protocol, or a null pointer if they don't find a match.

### Example

We'll see in Section 15.6 that when an application calls

```
socket(PF_INET, SOCK_STREAM, 0); /* TCP socket */
```

`pfindtype` gets called as

```
pfindtype(PF_INET, SOCK_STREAM);
```

Figure 7.12 shows that `pfindtype` will return a pointer to `inetsw[2]`, since TCP is the first `SOCK_STREAM` protocol in the array. Similarly,

```
socket(PF_INET, SOCK_DGRAM, 0); /* UDP socket */
```

leads to

```
pfindtype(PF_INET, SOCK_DGRAM);
```

which returns a pointer to UDP in `inetsw[1]`.

## 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).

```

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_protoswnPROTOSW; pr++)
150             if (pr->pr_ctlinput)
151                 (*pr->pr_ctlinput) (cmd, sa, (caddr_t) 0);
152 }
-----uipc_domain.c

```

Figure 7.21 `pfctlinput` function.

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.



## 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.

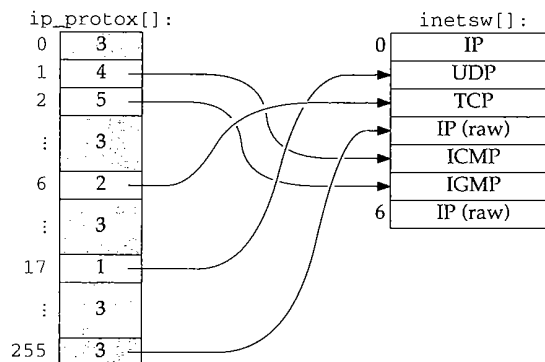


Figure 7.22 The `ip_protox` array maps the protocol number to an entry in the `inet_sw` array.

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 `inet_sw` array that processes the datagram. For example, a datagram with a protocol number of 6 is processed by `inet_sw[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 (`inet_sw[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 `inet_sw` (other than the entries with protocol numbers of 0 or

```

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 = ipqmaxlen;
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 }

```

Figure 7.23 ip\_init function.

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.

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 (`ipqmaxlen`) at compile time. A system administrator can use a kernel debugger such as `adb` to change `ipqmaxlen` 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.

## 7.9 sysctl System Call

The `sysctl` system call accesses and modifies Net/3 systemwide parameters. The system administrator can modify the parameters through the `sysctl(8)` program. Each parameter is identified by a hierarchical list of integers and has an associated type. The prototype for the system call is:

```
int sysctl(int *name, u_int namelen, void *old, size_t *oldlenp, void *new,
          size_t newlen);
```

`name` points to an array containing `namelen` integers. The old value is returned in the area pointed to by `oldp`, and the new value is passed in the area pointed to by `newp`.

Figure 7.24 summarizes the organization of the names related to networking.

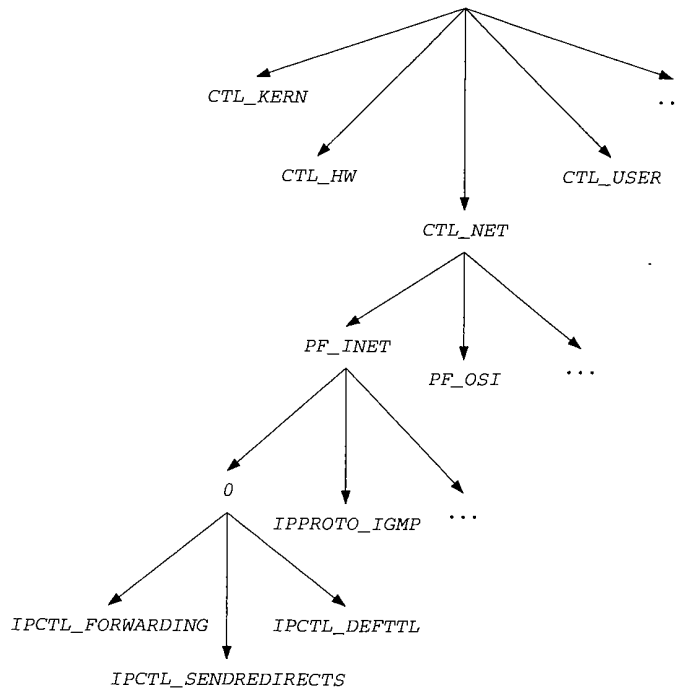


Figure 7.24 `sysctl` names.

In Figure 7.24, the full name for the IP forwarding flag would be

```
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 a different function. Figure 7.25 shows the functions that handle the Internet parameters.

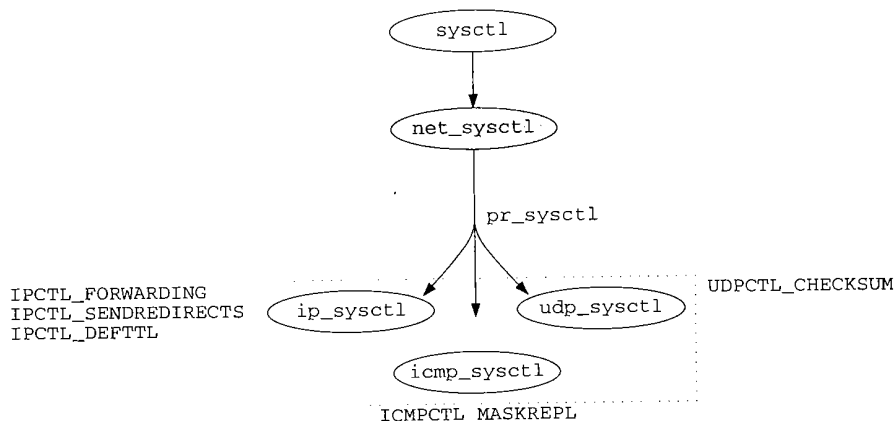


Figure 7.25 `sysctl` functions for Internet parameters.

The top-level names are processed by `sysctl`. The network-level names are processed by `net_sysctl`, which dispatches control based on the family and protocol to the `pr_sysctl` function specified in the protocol's `protosw` entry.

`sysctl` is implemented in the kernel by the `__sysctl` function, which we do not discuss in this text. It contains code to move the `sysctl` arguments to and from the kernel and a switch statement to select the appropriate function to process the arguments, in this case `net_sysctl`.

Figure 7.26 shows the `net_sysctl` function.

108-119 The arguments to `net_sysctl` are the same as those to the `sysctl` system call with the addition of `p`, which points to the current process structure.

120-134 The next two integers in the name are taken to be the protocol family and protocol numbers as specified in the domain and `protosw` structures. If no family is specified, 0 is returned. If a family is specified, the `for` loop searches the domain list for a matching family. `ENOPROTOOPT` is returned if a match is not found.

135-141 Within a matching domain, the second `for` loop locates the first matching protocol that has the `pr_sysctl` function defined. When a match is found, the request is passed to the `pr_sysctl` function for the protocol. Notice that name is advanced to pass the remaining integers down to the next level. If no matching protocol is found, `ENOPROTOOPT` is returned.

Figure 7.27 shows the `pr_sysctl` functions defined for the Internet protocols.

```

108 net_sysctl(name, namelen, oldp, oldlenp, newp, newlen, p)
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 (ENOPROTOPT);
137 found:
138     for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
139         if (pr->pr_protocol == protocol && pr->pr_sysctl)
140             return ((*pr->pr_sysctl) (name + 2, namelen - 2,
141                                     oldp, oldlenp, newp, newlen));
142     return (ENOPROTOPT);
143 }

```

uipc\_domain.c

Figure 7.26 net\_sysctl function.

pr_protocol	inetsw[]	pr_sysctl	Description	Reference
0	0	ip_sysctl	IP	Section 8.9
IPPROTO_UDP	1	udp_sysctl	UDP	Section 23.11
IPPROTO_ICMP	4	icmp_sysctl	ICMP	Section 11.14

Figure 7.27 pr\_sysctl functions for the Internet protocol family.

In the routing domain, pr\_sysctl points to the sysctl\_rtable function, which is described in Chapter 19.

## 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]`?

# 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.

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

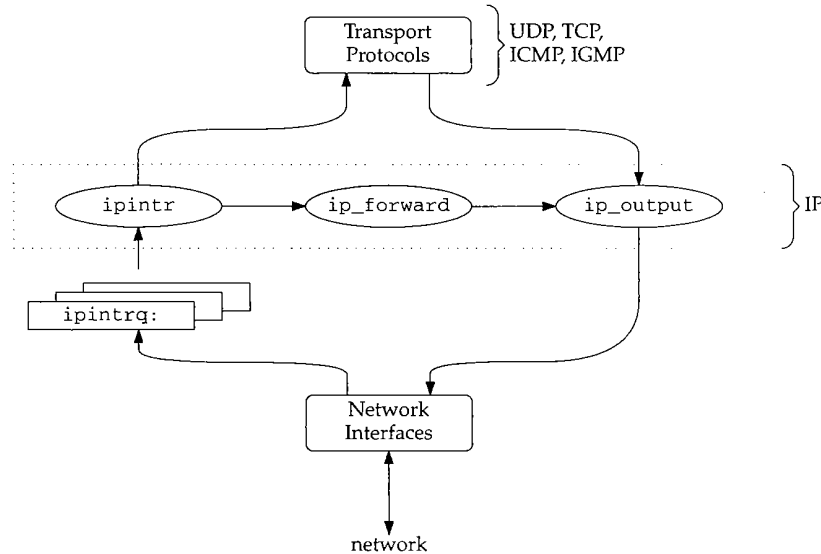


Figure 8.1 IP layer processing.

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.

## 8.2 Code Introduction

Two headers and three C files are 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

Figure 8.2 Files discussed in this chapter.

### Global Variables

Several global variables appear in the IP processing code. They are described in Figure 8.3.



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_protox	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

Figure 8.3 Global variables introduced in this chapter.

### 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.

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 8.4 Statistics collected in this chapter.

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

Figure 8.5 Sample IP statistics.

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.

SNMP variable	ipstat member	Description
ipDefaultTTL	ip_defttl	default TTL for datagrams (64 "hops")
ipForwarding	ipforwarding	is system acting as a router?
ipReasmTimeout	IPFRAGTTL	reassemble 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

Figure 8.6 Simple SNMP variables in IP group.

### 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.

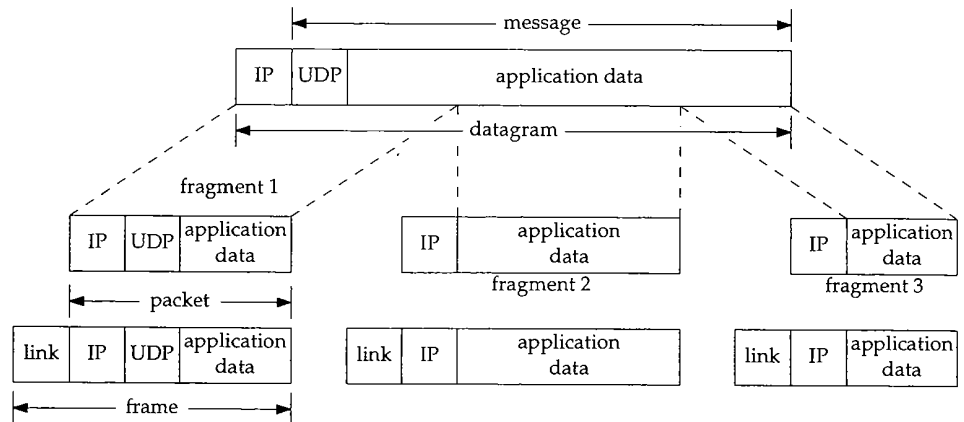


Figure 8.7 Frames, packets, fragments, datagrams, and messages.

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 includes the member names of the `ip` structure (shown in Figure 8.9) through which Net/3 accesses the IP header.

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.

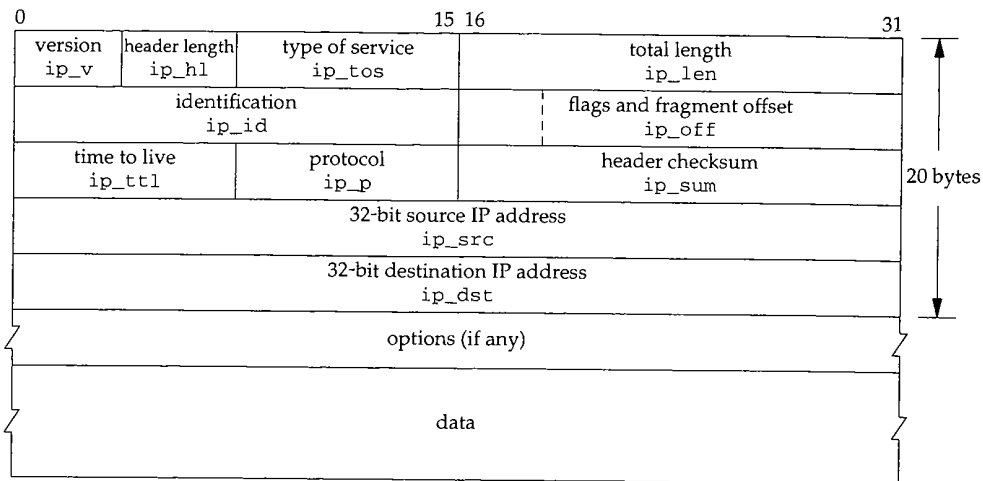


Figure 8.8 IP datagram, including the ip structure names.

```

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     u_char  ip_v:4;          /* version */
51 #endif
52 #if BYTE_ORDER == BIG_ENDIAN
53     u_char  ip_v:4,          /* version */
54     u_char  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_OFFMASK 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

Figure 8.9 ip structure.

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.

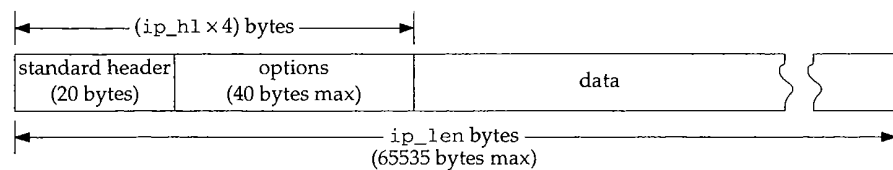


Figure 8.10 Organization of an IP packet with options.

Because `ip_hl` is measured in 4-byte units, IP options must always be padded to a 4-byte boundary.

## 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).
2. The SLIP interface handles only IP packets, so demultiplexing is unnecessary (Section 5.3).
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`.

```

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     next:
109     /*
110      * Get next datagram off input queue and get IP header
111      * in first mbuf.
112      */
113     s = splimp();
114     IF_DEQUEUE(&ipintrq, m);
115     splx(s);
116     if (m == 0)
117         return;

118     /* input packet processing */
119     /* Figures 8.12, 8.13, 8.15, 10.11, and 12.40 */

332     goto next;
333     bad:
334     m_freem(m);
335     goto next;
336 }

```

Figure 8.11 `ipintr` function.

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 `sinput` 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.

```

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*

Figure 8.12 ipintr function.

**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.

```

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_freem(m);
268     } else
269         ip_forward(m, 0);
270     goto next;
271 ours:

```

ip\_input.c

Figure 8.13 ipintr continued.

### 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),
- 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.


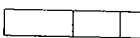






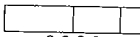
Variable	Ethernet	SLIP	Loopback	Lines (Figure 8.13)
<code>ia_addr</code>	 140.252.13.33	 140.252.1.29	 127.0.0.1	192-193
<code>ia_broadaddr</code>	 140.252.13.63			198-200
<code>ia_netbroadcast</code>	 140.252.255.255			201-202
<code>ia_subnet</code>	 140.252.13.32			207-209
<code>ia_net</code>	 140.252.0.0			210-211
<code>INADDR_BROADCAST</code>		 255.255.255.255		258-259
<code>INADDR_ANY</code>		 0.0.0.0		260-261

Figure 8.14 Comparisons to determine whether or not a packet has reached its final destination.

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.

```
----- 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
```

Figure 8.15 `ipintr` continued.

### 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.

## 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

```
----- route.h
46 struct route {
47     struct rtable *ro_rt; /* pointer to struct with information */
48     struct sockaddr ro_dst; /* destination of this route */
49 };
----- route.h
```

Figure 8.16 `route` structure.

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?

7-871 The first argument to `ip_forward` is a pointer to an mbuf chain containing the packet to be forwarded. If the second argument, `srct`, is nonzero, the packet is being forwarded because of a source route option (Section 9.6).

7-884 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.

```

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_freem(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             RTFREE(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]++;

```

Figure 8.17 ip\_forward function: route selection.



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

185-890 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

991-907 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.

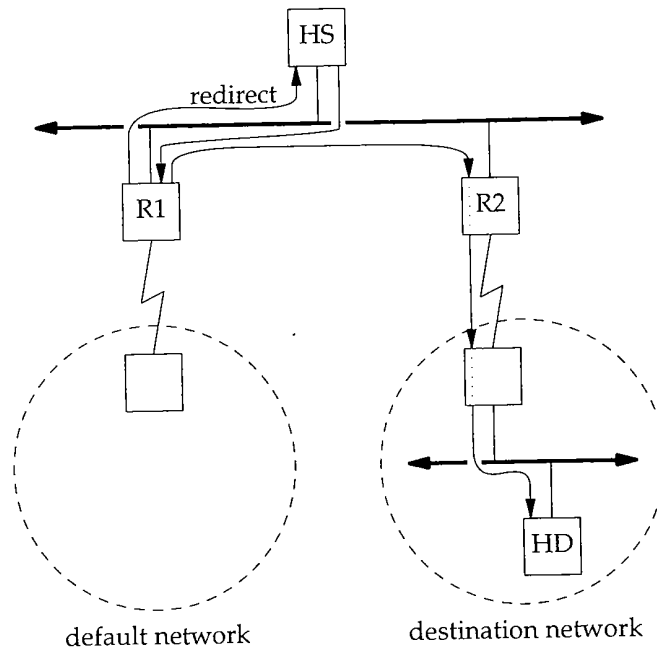


Figure 8.18 Router R1 is redirecting host HS to use router R2 to reach HD.

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.

#### 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.

```

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
931     if (RTA(rt) &&
932         (src & RTA(rt)->ia_subnetmask) == RTA(rt)->ia_subnet) {
933         if (rt->rt_flags & RTF_GATEWAY)
934             dest = satosin(rt->rt_gateway)->sin_addr.s_addr;
935         else
936             dest = ip->ip_dst.s_addr;
937         /* Router requirements says to only send host redirects */
938         type = ICMP_REDIRECT;
939         code = ICMP_REDIRECT_HOST;
940     }

```

Figure 8.19 ip\_forward continued.

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.

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.

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.

```
941 error = ip_output(m, (struct mbuf *) 0, &ipforward_rt, ip_input.c
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 case 0:
961     /* forwarded, but need redirect */
962     /* type, code set above */
963     break;
964
965 case ENETUNREACH:
966     /* shouldn't happen, checked above */
967 case EHOSTUNREACH:
968 case ENETDOWN:
969 case EHOSTDOWN:
970 default:
971     type = ICMP_UNREACH;
972     code = ICMP_UNREACH_HOST;
973     break;
974
975 case EMSGSIZE:
976     type = ICMP_UNREACH;
977     code = ICMP_UNREACH_NEEDFRAG;
978     if (ipforward_rt.ro_rt)
979         destifp = ipforward_rt.ro_rt->rt_ifp;
980     ipstat.ips_cantfrag++;
981     break;
982
983 case ENOBUFS:
984     type = ICMP_SOURCEQUENCH;
985     code = 0;
986     break;
987 }
988 icmp_error(mcopy, type, code, dest, destifp);
989 }
```

Figure 8.20 ip\_forward continued.

ip\_input.c

Error code from <code>ip_output</code>	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  EHOSTDOWN default	ICMP_UNREACH_HOST	A route to the host could not be found. The outgoing interface specified by the route is not operating. The interface could not send the packet to the selected host. Any unrecognized error is reported as an ICMP_UNREACH_HOST error.

Figure 8.21 Errors from `ip_output`.

## 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`).

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.

`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.

```

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     }

```

Figure 8.22 ip\_output function.

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.

Figure 8.23 ip\_output: flags values.

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 header—length, 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.

```

----- 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 rtable *) 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            rtalloc(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*

Figure 8.24 ip\_output continued.

**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.

```

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      */

```

ip\_output.c

```

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
241     sendit:
242     /*
243     * If small enough for interface, can just send directly.
244     */
245     if ((u_short) ip->ip_len <= ifp->if_mtu) {
246         ip->ip_len = htons((u_short) ip->ip_len);
247         ip->ip_off = htons((u_short) ip->ip_off);
248         ip->ip_sum = 0;
249         ip->ip_sum = in_cksum(m, hlen);
250         error = (*ifp->if_output) (ifp, m,
251                                 (struct sockaddr *) dst, ro->ro_rt);
252     }
253
254     /* fragmentation (Section 10.3) */
255
256     done:
257     if (ro == &iproute && (flags & IP_ROUTETOIF) == 0 && ro->ro_rt)
258         RTFREE(ro->ro_rt);
259     return (error);
260 bad:
261     m_freem(m0);
262     goto done;
263 }

```

*ip\_output.c*

Figure 8.25 ip\_output continued.

**Select source address**

2-239 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.

## 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`.

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

Figure 8.26 in\_cksum versions in Net/3.

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.
2. 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.
3. 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:

```

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 }

```

Figure 8.27 A naive implementation of the IP checksum calculation.

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.

```
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 ((1 & (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     }
```

```

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

Figure 8.28 An optimized portable C implementation of the IP checksum calculation.



## 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`.

## 8.8 setsockopt and getsockopt System Calls

Net/3 provides access to several networking features through the `setsockopt` and `getsockopt` system calls. These system calls support a generic interface used by a process to access features of a networking protocol that aren't supported by the standard system calls. The prototypes for these two calls are:

```
int setsockopt(int s, int level, int optname, const void *optval, int optlen);  
  
int getsockopt(int s, int level, int optname, void *optval, int *optlen);
```

Most socket options affect only the socket on which they are issued. Compare this to `sysctl` parameters, which affect the entire system. The socket options associated with multicasting are a notable exception and are described in Chapter 12.

`setsockopt` and `getsockopt` set and get options at all levels of the communication stack. Net/3 processes options according to the protocol associated with `s` and the identifier specified by `level`. Figure 8.29 lists possible values for `level` within the protocols that we discuss.

We describe the implementation of the `setsockopt` and `getsockopt` system calls in Chapter 17, but we discuss the implementation of individual options within the

Domain	Protocol	level	Function	Reference
any	any	<i>SOL_SOCKET</i>	so <code>setopt</code> and so <code>getopt</code>	Figures 17.5 and 17.11
IP	UDP	<i>IPPROTO_IP</i>	ip <code>ctloutput</code>	Figure 8.31
	TCP	<i>IPPROTO_TCP</i>	tcp <code>ctloutput</code>	Section 30.6
		<i>IPPROTO_IP</i>	ip <code>ctloutput</code>	Figure 8.31
raw IP ICMP IGMP		<i>IPPROTO_IP</i>	rip <code>ctloutput</code> and ip <code>ctloutput</code>	Section 32.8

Figure 8.29 `setsockopt` and `getsockopt` arguments.

optname	optval type	Function	Description
<i>IP_OPTIONS</i>	void *	in <code>pcbopts</code>	set or get IP options to be included in outgoing datagrams
<i>IP_TOS</i>	int	ip <code>ctloutput</code>	set or get IP TOS for outgoing datagrams
<i>IP_TTL</i>	int	ip <code>ctloutput</code>	set or get IP TTL for outgoing datagrams
<i>IP_RECVDSTADDR</i>	int	ip <code>ctloutput</code>	enable or disable queueing of IP destination address (UDP only)
<i>IP_RECVOPTS</i>	int	ip <code>ctloutput</code>	enable or disable queueing of incoming IP options as control information (UDP only, not implemented)
<i>IP_RECVRETOPTS</i>	int	ip <code>ctloutput</code>	enable or disable queueing of reversed source route associated with incoming datagram (UDP only, not implemented)

Figure 8.30 Socket options: *IPPROTO\_IP* level for *SOCK\_RAW*, *SOCK\_DGRAM*, or *SOCK\_STREAM* sockets.

appropriate chapters. In this chapter, we cover the options that provide access to IP features.

Throughout the text we summarize socket options as shown in Figure 8.30. This figure shows the options for the *IPPROTO\_IP* level. The option appears in the first column, the data type of the variable pointed to by *optval* appears in the second column, and the third column shows the function that processes the option.

Figure 8.31 shows the overall organization of the `ip_ctloutput` function, which handles most of the *IPPROTO\_IP* options. In Section 32.8 we show the additional *IPPROTO\_IP* options that work with *SOCK\_RAW* sockets.

431-447 `ip_ctloutput`'s first argument, *op*, is either `PRCO_SETOPT` or `PRCO_GETOPT`. The second argument, *so*, points to the socket on which the request was issued. *level* must be *IPPROTO\_IP*. *optname* is the option to change or to retrieve, and *mp* points indirectly to an mbuf that contains the related data for the option. *m* is initialized to point to the mbuf referenced by *\*mp*.

448-500 If an unrecognized option is specified in the call to `setsockopt` (and therefore to the `PRCO_SETOPT` case of the switch), `ip_ctloutput` releases any mbuf passed by the caller and returns `EINVAL`.

```

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
443     if (level != IPPROTO_IP) {
444         error = EINVAL;
445         if (op == PRCO_SETOPT && *mp)
446             (void) m_free(*mp);
447     } else
448         switch (op) {
449             case PRCO_SETOPT:
450                 switch (optname) {
451
452                     /* PRCO_SETOPT processing (Figures 8.32 and 12.17) */
453
454                     freeit:
455                     default:
456                         error = EINVAL;
457                         break;
458                     }
459                     if (m)
460                         (void) m_free(m);
461                     break;
462
463             case PRCO_GETOPT:
464                 switch (optname) {
465
466                     /* PRCO_GETOPT processing (Figures 8.33 and 12.17) */
467
468                     default:
469                         error = ENOPROTOOPT;
470                         break;
471                     }
472                 break;
473             }
474     return (error);
475 }

```

Figure 8.31 ip\_ctloutput function: overview.

501-553 Unrecognized options passed to getsockopt result in ip\_ctloutput returning ENOPROTOOPT. In this case, the caller releases the mbuf.

**PRCO\_SETOPT Processing**

The processing for PRCO\_SETOPT is shown in Figure 8.32.

```

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;

```

Figure 8.32 ip\_ctloutput function: PRCO\_SETOPT processing.

450-451 IP\_OPTIONS is processed by ip\_pcbopts (Figure 9.32).

452-484 The IP\_TOS, IP\_TTL, IP\_RECVOPTS, IP\_RECVRETOPTS, and IP\_RECVDSTADDR options all expect an integer to be available in the mbuf pointed to by m. The integer is stored in optval and then used to change the ip\_tos or ip\_ttl values associated with the socket or to set or clear the INP\_RECVOPTS, INP\_RECVRETOPTS, or INP\_RECVDSTADDR flags associated with the socket. The macro OPTSET sets (or clears) the specified bit if optval is nonzero (or 0).

Figure 8.30 showed that `IP_RECVOPTS` and `IP_RECVRETOPTS` were not implemented. In Chapter 23, we'll see that the settings of these options are ignored by UDP.

### PRCO\_GETOPT Processing

Figure 8.33 shows the code that retrieves the IP options when `PRCO_GETOPT` is specified.

```

503         case IP_OPTIONS:                                ip_output.c
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         case IP_TOS:
513         case IP_TTL:
514         case IP_RECVOPTS:
515         case IP_RECVRETOPTS:
516         case IP_RECVDSTADDR:
517             *mp = m = m_get(M_WAIT, MT_SOOPTS);
518             m->m_len = sizeof(int);
519             switch (optname) {
520                 case IP_TOS:
521                     optval = inp->inp_ip.ip_tos;
522                     break;
523                 case IP_TTL:
524                     optval = inp->inp_ip.ip_ttl;
525                     break;
526 #define OPTBIT(bit) (inp->inp_flags & bit ? 1 : 0)
527                 case IP_RECVOPTS:
528                     optval = OPTBIT(INP_RECVOPTS);
529                     break;
530                 case IP_RECVRETOPTS:
531                     optval = OPTBIT(INP_RECVRETOPTS);
532                     break;
533                 case IP_RECVDSTADDR:
534                     optval = OPTBIT(INP_RECVDSTADDR);
535                     break;
536             }
537             *mtod(m, int *) = optval;
538             break;

```

Figure 8.33 `ip_ctloutput` function: `PRCO_GETOPT` processing.

503-538 For `IP_OPTIONS`, `ip_ctloutput` returns an mbuf containing a copy of the options associated with the socket. For the remaining options, `ip_ctloutput` returns

the value of `ip_tos`, `ip_ttl`, or the state of the flag associated with the option. The value is returned in the mbuf pointed to by `m`. The macro `OPTBIT` returns 1 (or 0) if bit is on (or off) in `inp_flags`.

Notice that the IP options are stored in the protocol control block (`inp`, Chapter 22) associated with the socket.

### 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`.

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 8.34 `ip_sysctl` parameters.

Figure 8.35 shows the `ip_sysctl` function.

```

184 int
185 ip_sysctl(name, namelen, oldp, oldlenp, newp, newlen)
186 {
187     int *name;
188     int namelen;
189     void *oldp;
190     size_t *oldlenp;
191     void *newp;
192     size_t newlen;
193 {
194     /* All sysctl names at this level are terminal. */
195     if (namelen != 1)
196         return (ENOTDIR);
197
198     switch (name[0]) {
199     case IPCTL_FORWARDING:
200         return (sysctl_int(oldp, oldlenp, newp, newlen, &ipforwarding));
201     case IPCTL_SENDREDIRECTS:
202         return (sysctl_int(oldp, oldlenp, newp, newlen,
203                             &ipsendredirects));
204     case IPCTL_DEFTTL:
205         return (sysctl_int(oldp, oldlenp, newp, newlen, &ip_defttl));
206     default:
207         return (EOPNOTSUPP);
208     }
209     /* NOTREACHED */
210 }

```

*ip\_input.c*

*ip\_input.c*

Figure 8.35 `ip_sysctl` function.

- 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.

## 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

- 8.1 Should IP accept broadcast packets 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?
- 8.5 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 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?

- 8.6 A new host is attached to a subnetted network and is configured to perform routing (`ipforwarding` equals 1) but its network interface has not been assigned a subnet mask. What happens when this host receives a subnet broadcast packet?
- 8.7 Why is it necessary to decrement `ip_ttl` after testing it (versus before) in Figure 8.17?
- 8.8 What would happen if two routers each considered the other the best next-hop destination for a packet?
- 8.9 Which addresses would not be checked in Figure 8.14 for a packet arriving at the SLIP interface? Would any additional addresses be checked that aren't listed in Figure 8.14?
- 8.10 `ip_forward` converts the fragment id from host byte order to network byte order before calling `icmp_error`. Why does it not also convert the fragment offset?



# 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.

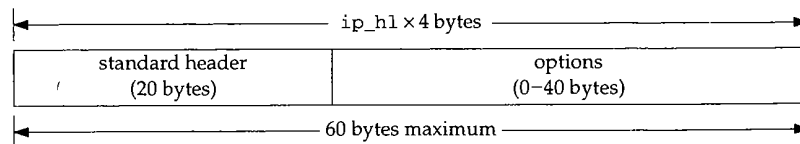


Figure 9.1 An IP header may contain 0 to 40 bytes of IP options.

### 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.

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

Figure 9.2 Files discussed in this chapter.

### Global Variables

The two global variables described in Figure 9.3 support the reversal of source routes.

Variable	Datatype	Description
ip_nhops	int	hop count for previous source route
ip_srort	struct ip_srort	previous source route

Figure 9.3 Global variables introduced in this chapter.

### Statistics

The only statistic updated by the options processing code is `ips_badoptions` from the `ipstat` structure, which Figure 8.4 described.

## 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.

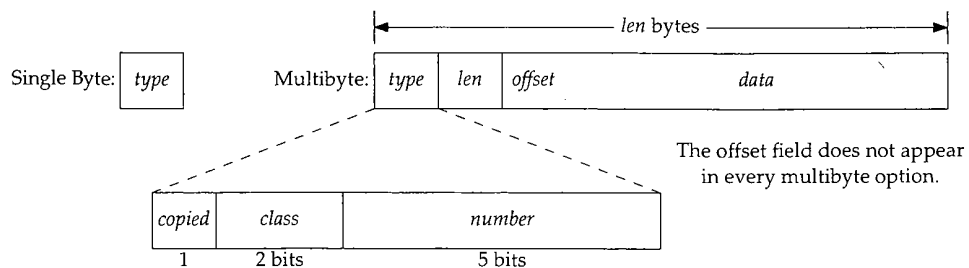


Figure 9.4 The organization of single-byte and multibyte IP options.

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.

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)
	1-0-5 133	1-00-00101	varies		extended security
<i>IPOPT_SATID</i>	1-0-8 136	1-00-01000	4		stream identifier
<i>IPOPT_SSRR</i>	1-0-9 137	1-00-01001	varies	•	strict source and record route (SSRR)

Figure 9.5 IP options defined by RFC 791.

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.

<i>class</i>	Description
0	control
1	reserved
2	debugging and measurement
3	reserved

Figure 9.6 The *class* field within an IP option.

## 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`.

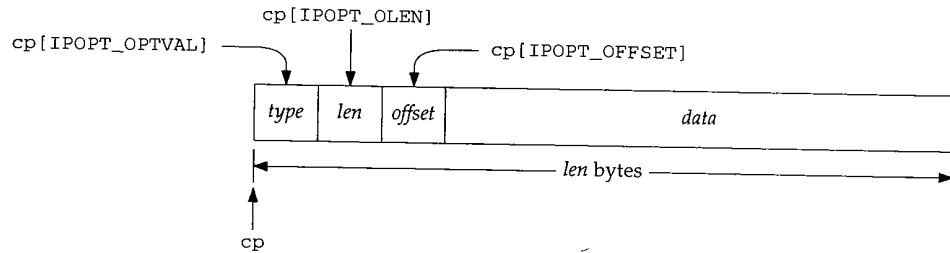


Figure 9.7 Access to IP option fields is by constant offsets.

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.

553-566 `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.

```

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     dst = ip->ip_dst;
565     cp = (u_char *) (ip + 1);
566     cnt = (ip->ip_hl << 2) - sizeof(struct ip);
567     for (; cnt > 0; cnt -= optlen, cp += optlen) {
568         opt = cp[IPOPT_OPTVAL];
569         if (opt == IPOPT_EOL)
570             break;
571         if (opt == IPOPT_NOP)
572             optlen = 1;
573         else {
574             optlen = cp[IPOPT_OLEN];
575             if (optlen <= 0 || optlen > cnt) {
576                 code = &cp[IPOPT_OLEN] - (u_char *) ip;
577                 goto bad;
578             }
579         }
580         switch (opt) {

581             default:
582                 break;

583                 /* option processing */

584         }

585     }

586     if (forward) {
587         ip_forward(m, 1);
588         return (1);
589     }

590     return (0);

591 bad:
592     ip->ip_len -= ip->ip_hl << 2; /* XXX icmp_error adds in hdr length */
593     icmp_error(m, type, code, 0, 0);
594     ipstat.ips_badoptions++;
595     return (1);
596 }

```

Figure 9.8 ip\_dooptions function.

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_doptions` 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_doptions` 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_doptions` returns 1 to prevent `ipintr` from processing the discarded packet.

The following sections describe each of the options that are processed by Net/3.

## 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.

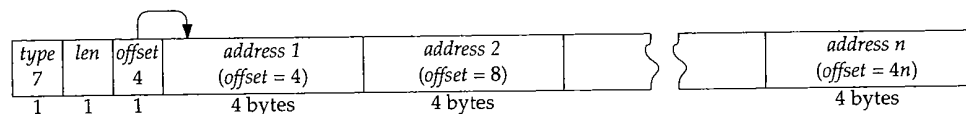


Figure 9.9 The record route option.  $n$  must be  $\leq 9$ .

647-657 If the option offset is too small, `ip_doptions` 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

```

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--;
656             /* 0 origin */
657             if (off > optlen - sizeof(struct in_addr))
658                 break;
659             bcopy((caddr_t) (&ip->ip_dst), (caddr_t) &ipaddr.sin_addr,
660                 sizeof(ipaddr.sin_addr));
661             /*
662              * locate outgoing interface; if we're the destination,
663              * use the incoming interface (should be same).
664              */
665             if ((ia = (INA) ifa_ifwithaddr((SA) &ipaddr)) == 0 &&
666                 (ia = ip_rtaddr(ipaddr.sin_addr)) == 0) {
667                 type = ICMP_UNREACH;
668                 code = ICMP_UNREACH_HOST;
669                 goto bad;
670             }
671             bcopy((caddr_t) &(IA_SIN(ia)->sin_addr),
672                 (caddr_t) (cp + off), sizeof(struct in_addr));
673             cp[IPOPT_OFFSET] += sizeof(struct in_addr);
674             break;

```

Figure 9.10 `ip_dooptions` function: record route option processing.

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.

#### 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.

```

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 }

```

Figure 9.11 ip\_rtaddr function: locate 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.

## 9.6 Source and Record Route Options

Normally a packet is forwarded along a path chosen by the intermediate routers. The source and record route options allow the source host to specify an explicit path to the destination that overrides routing decisions of the intermediate routers. Furthermore, the route is recorded as the packet travels toward its destination.

A *strict* route includes the address of every intermediate router between the source and destination; a *loose* route specifies only some of the intermediate routers. Routers are free to choose any path between two systems listed in a loose route, whereas no intermediate routers are allowed between the systems listed in a strict route. We'll use Figure 9.12 to illustrate source route processing.

A, B, and C are routers and HS and HD are the source and destination hosts. Since each interface has its own IP address, we see that router A has three addresses:  $A_1$ ,  $A_2$ ,



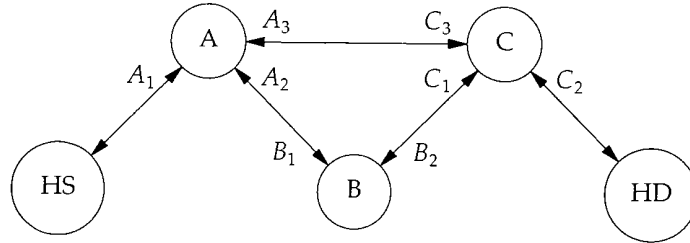


Figure 9.12 Source route example.

and  $A_3$ . Similarly, routers B and C have multiple addresses. Figure 9.13 shows the format of the source and record route options.

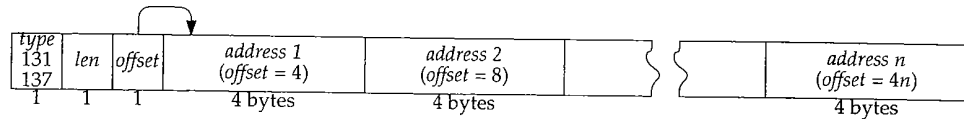


Figure 9.13 The loose and strict source routing options.

The source and destination addresses in the IP header and the offset and address list in the option specify the route and the packet's current location within the route. Figure 9.14 shows how this information changes as the packet follows the loose source route from HS to A to B to C to HD. The loose source route specified by the process are the four IP addresses:  $A_3$ ,  $B_1$ ,  $C_1$ , and HD. Each row represents the state of the packet when sent by the system shown in the first column. The last line shows the packet as received by HD. Figure 9.15 shows the relevant code.

System	IP Header		Source Route Option			
	ip_src	ip_dst	offset	addresses		
HS	HS	$A_3$	4	• $B_1$	$C_1$	HD
A	HS	$B_1$	8	$A_2$	• $C_1$	HD
B	HS	$C_1$	12	$A_2$	$B_2$	• HD
C	HS	HD	16	$A_2$	$B_2$	$C_2$ •
HD	HS	HD	16	$A_2$	$B_2$	$C_2$ •

Figure 9.14 The source route option is modified as a packet traverses the route.

The • marks the position of *offset* relative to the addresses within the route. Notice that the address of the outgoing interface is placed in the option by each system. In particular, the original route specified  $A_3$  as the first-hop destination but the output interface,  $A_2$ , was recorded in the route. In this way, the route taken by the packet is recorded in the option. This recorded route should be reversed by the destination system and attached to any reply packets so that they follow the same path as the initial packet but in the reverse direction.

Except for UDP, Net/3 reverses a received source route when responding.

```

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;

```

input.c

```

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

Figure 9.15 ip\_dooptions function: LSRR and SSRR option processing.

583-612 Net/3 sends an ICMP parameter problem error with the appropriate value of code if the option offset is smaller than 4 (IPOPT\_MINOFF). If the destination address of the packet does not match one of the local addresses and the option is a strict source route (IPOPT\_SSRR), an ICMP source route failure error is sent. If a local address isn't listed in the route, the previous system sent the packet to the wrong host. This isn't an error for a loose source route (IPOPT\_LSRR); it means IP must forward the packet toward the destination.

**End of source route**

613-620 Decrementing *off* converts it to a byte offset from the start of the option. If *ip\_dst* in the IP header is one of the local addresses and *off* points beyond the end of the source route, there are no more addresses in the source route and the packet has reached its final destination. *save\_rte* makes a copy of the route in the static structure *ip\_srcrt* and saves the number of addresses in the route in the global *ip\_nhops* (Figure 9.18).

*ip\_srcrt* is declared as an external static structure since it is only accessed by the functions declared in *ip\_input.c*.

**Update packet for next hop**

621-637 If *ip\_dst* is one of the local addresses and *offset* points to an address within the option, this system is an intermediate system specified in the source route and the packet has not reached its final destination. During strict routing, the next system must be on a directly connected network. *ifa\_ifwithdst* and *ifa\_ifwithnet* locate a route to the next system by searching the configured interfaces for a matching destination address (a point-to-point interface) or a matching network address (a broadcast interface). During loose routing, *ip\_rtaddr* (Figure 9.11) locates the route to the next system by querying the routing tables. If no interface or route is found for the next system, an ICMP source route failure error is sent.

638-644 If an interface or a route is located, *ip\_dooptions* sets *ip\_dst* to the IP address pointed to by *off*. Within the source route option, the intermediate address is replaced with the address of the outgoing interface, and the offset is incremented to point to the next address in the route.

**Multicast destinations**

645-646 If the new destination address is not a multicast address, setting forward to 1 indicates that the packet should be forwarded after `ip_dooptions` processes all the options instead of returning the packet to `ipintr`.

Multicast addresses within a source route enable two multicast routers to communicate through intermediate routers that don't support multicasting. Chapter 14 describes this technique in more detail.

Section 8.5 of Volume 1 contains more examples of the source route options.

**save\_rte Function**

RFC 1122 requires that the route recorded in a packet be made available to the transport protocol at the final destination. The transport protocols must reverse the route and attach it to any reply packets. The function `save_rte`, shown in Figure 9.18, saves source routes in an `ip_srcrt` structure, shown in Figure 9.16

```

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;

```

*ip\_input.c*

Figure 9.16 `ip_srcrt` structure.

The declaration of `route` is incorrect, though the error is benign. It should be

```
struct in_addr route[(MAX_IPOPTLEN - 3) / sizeof(struct in_addr)];
```

The discussion with Figures 9.26 and 9.27 covers this in more detail.

57-63 This code defines the `ip_srcrt` structure and declares the static variable `ip_srcrt`. Only two functions access `ip_srcrt`: `save_rte`, which copies the source route from an incoming packet into `ip_srcrt`; and `ip_srcroute`, which creates a reversed source route from `ip_srcrt`. Figure 9.17 illustrates source route processing.

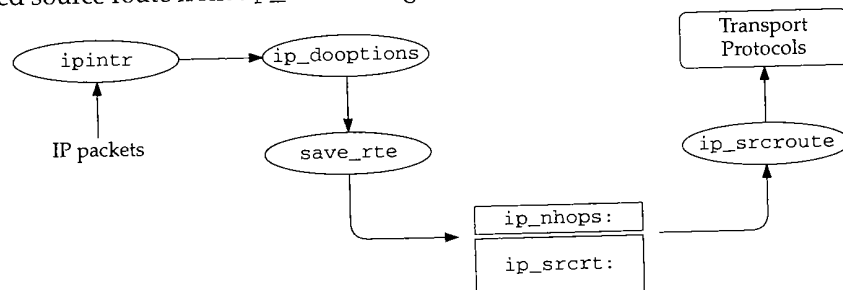


Figure 9.17 Processing of reversed source routes.

```

759 void
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 }

```

Figure 9.18 save\_rte function.

759-771 ip\_dooptions calls save\_rte when a source routed packet has reached its final destination. option is a pointer to a packet's source route option, and dst is ip\_src from the packet's header (i.e., the destination of the return route, HS from Figure 9.12). If the option length is larger than the ip\_srcrt structure, save\_rte returns immediately.

This would never happen, as the ip\_srcrt structure is larger than the largest option length (40 bytes).

save\_rte copies the option into ip\_srcrt, computes and saves the number of hops in the source route in ip\_nhops, and saves the destination of the return route in dst.

### ip\_srcroute Function

When responding to a packet, ICMP and the standard transport protocols must reverse any source route that the packet carried. The reversed source route is constructed from the saved route by ip\_srcroute, which is shown in Figure 9.19.

777-783 ip\_srcroute reverses the route saved in the ip\_srcrt structure and returns the result formatted as an ipoption structure (Figure 9.26). If ip\_nhops is 0, there is no saved route, so ip\_srcroute returns a null pointer.

Recall that in Figure 8.13, ipintr cleared ip\_nhops when a valid packet arrives. The transport protocols must call ip\_srcroute and save the reversed route themselves before the next packet arrives. As noted earlier, this is OK since the transport layer (TCP or UDP) is called by ipintr for each packet, before the next packet on IP's input queue is processed.

### Allocate mbuf for source route

784-790 If ip\_nhops is nonzero, ip\_srcroute allocates an mbuf and sets m\_len large enough to include the first-hop destination, the option header information (OPTSIZ), and the reversed route. If the allocation fails, a null pointer is returned as if there were no source route available.

```

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 + srcopt 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     * Record return path as an IP source route,
814     * reversing the path (pointers are now aligned).
815     */
816     while (p >= ip_srcrt.route) {
817         *q++ = *p--;
818     }
819
820     /*
821     * Last hop goes to final destination.
822     */
823     *q = ip_srcrt.dst;
824     return (m);
825 }

```

Figure 9.19 ip\_srcroute function.

791-804 p is initialized to point to the end of the incoming route, and ip\_srcroute copies the last recorded address to the front of the mbuf where it becomes the outgoing first-hop destination for the reversed route. Then the function copies a NOP option (Exercise 9.4) and the source route information into the mbuf.

ip\_input.c

805-818 The while loop copies the remaining IP addresses from the source route into the mbuf in reverse order. The last address in the route is set to the source address from the incoming packet, which save\_rte placed in ip\_srcrt.dst. A pointer to the mbuf is returned. Figure 9.20 illustrates the construction of the reversed route with the route from Figure 9.12.

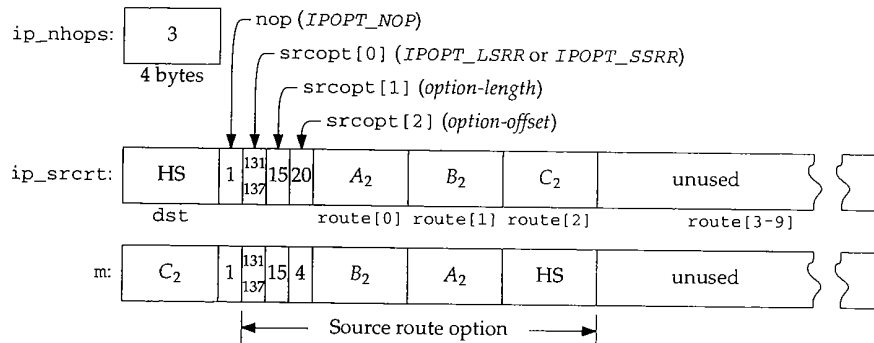


Figure 9.20 ip\_srcroute reverses the route in ip\_srcrt.

## 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`.

ipt_flg	Value	Description
<code>IPOPT_TS_TSONLY</code>	0	record timestamps
<code>IPOPT_TS_TSANDADDR</code>	1	record addresses and timestamps
	2	reserved
<code>IPOPT_TS_PRESPEC</code>	3	record timestamps only at the prespecified systems
	4-15	reserved

Figure 9.21 Possible values for `ipt_flg`.

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

\*/  
addr) +

- ip\_input.c

te copies  
oing first-  
ion (Exer-

```

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     u_char  ipt_oflw:4;        /* overflow counter */
121 #endif
122 #if BYTE_ORDER == BIG_ENDIAN
123     u_char  ipt_oflw:4;        /* overflow counter */
124     u_char  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 9.22 ip\_timestamp structure and constants.

ipt\_flg of 3, the originating host fills in the addresses of the systems at which a timestamp should be recorded when it constructs the option. Figure 9.23 shows the organization of the three timestamp options.

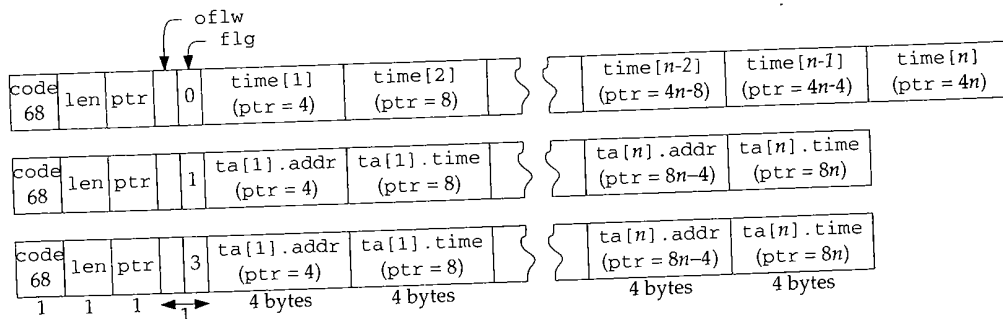


Figure 9.23 The three timestamp options (ipt\_ omitted).

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.

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.



—ip.h

—ip.h

time-  
organi-

e[n]  
= 4n

limited  
stamps  
at time-

h is less  
e num-  
ion was  
16 (it is

```

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         ntime = iptime();
719         bcopy((caddr_t) & ntime, (caddr_t) cp + ipt->ipt_ptr - 1,
720             sizeof(n_time));
721         ipt->ipt_ptr += sizeof(n_time);
722     }

```

Figure 9.24 ip\_dooptions function: timestamp option processing.

**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`.

```

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 }

```

ip\_icmp.c

ip\_icmp.c

Figure 9.25 `iptime` function.

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.

### 9.8 ip\_insetoptions 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\_insetoptions (called by ip\_output in Figure 8.22).

ip\_insetoptions expects the options to be formatted in an ipoption structure, shown in Figure 9.26.

```

92 struct ipoption {
93     struct in_addr ipopt_dst; /* first-hop dst if source routed */
94     char ipopt_list[MAX_IPOPTLEN]; /* options proper */
95 };

```

ip\_var.h

Figure 9.26 ipoption structure.

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.

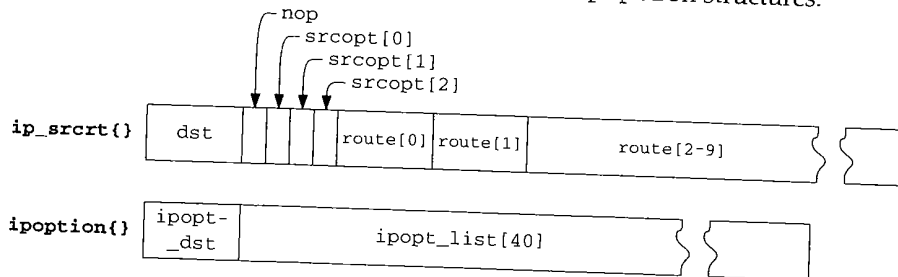


Figure 9.27 The ip\_srcrt and ipoption structures.

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).

The ip\_insetoptions function is shown in Figure 9.28. ip\_insetoptions 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\_insetoptions 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.

```

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 }

```

Figure 9.28 ip\_insertoptions function.

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.

out.c

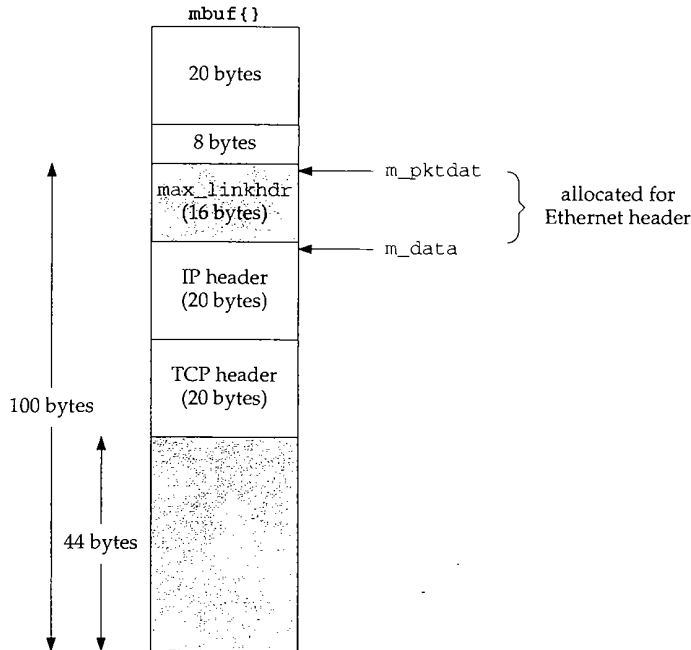


Figure 9.29 ip\_insertoptions function: TCP segment.

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.

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.

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.

en);

output.c

lude a  
estima-

for the  
segment

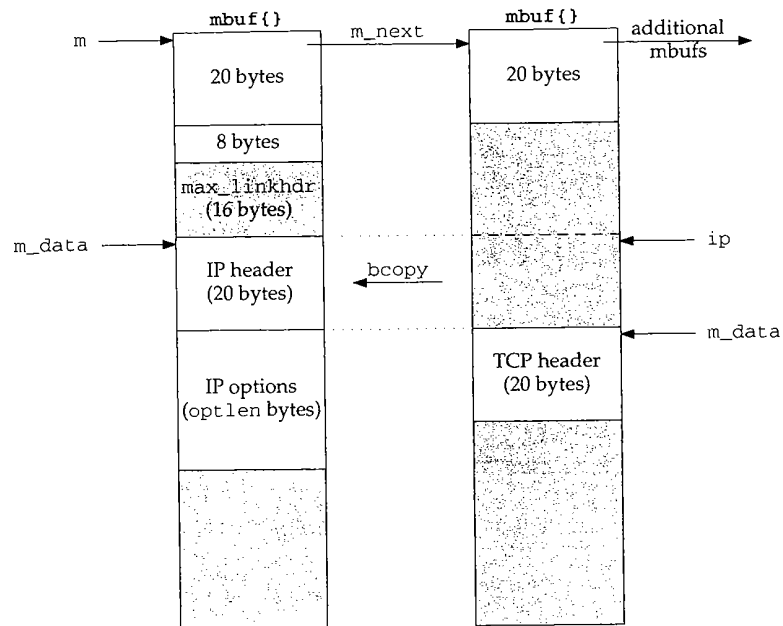


Figure 9.30 ip\_insertoptions function: TCP segment, after options have been copied.

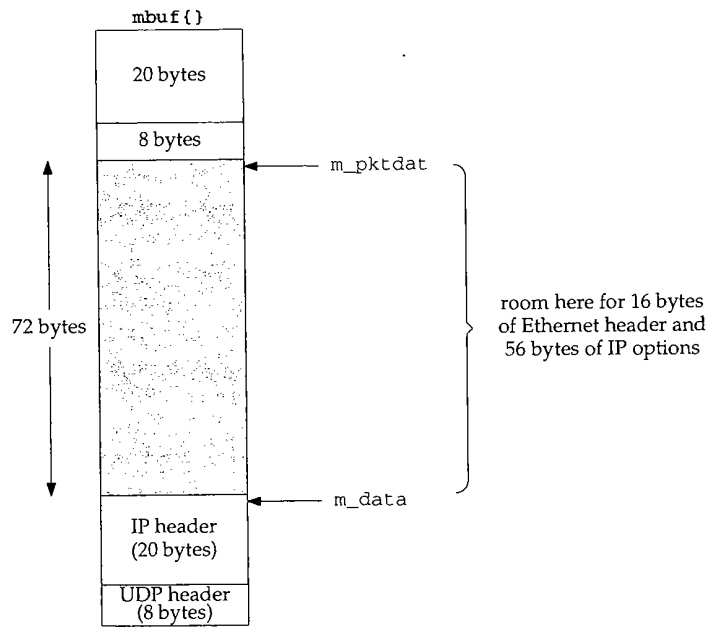


Figure 9.31 ip\_insertoptions function: UDP datagram.

## 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.

```
-----ip_output.c
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     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*

Figure 9.32 ip\_pcbopts function.

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.

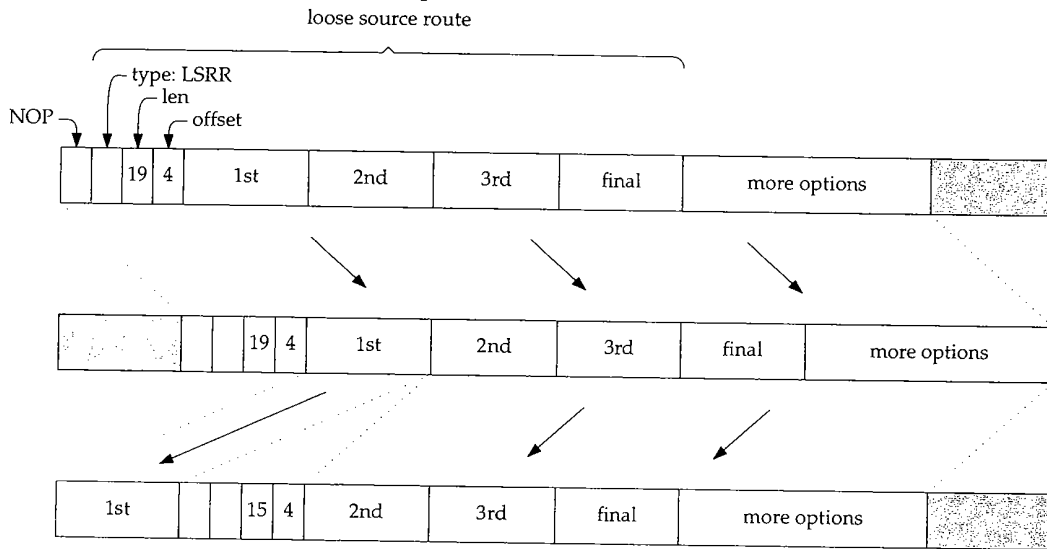


Figure 9.33 ip\_pcbopts option list processing.

**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 multi-byte 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.

output.c

ptions.  
ructed  
m. The  
option  
l SSRR  
ress in

### 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.

## 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 `tracert`. 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.

## 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

- 9.1 What would happen if a packet contained two different source route options?
- 9.2 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 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.
- 9.3 Some hosts may not be configured with a default route. In general, this prevents communication with the host 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?
- 9.5 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?

e no  
cket

itive  
and  
ram-  
sup-  
ecify

large  
hops  
n the  
with a

ents,  
tions  
ation  
ven if

er the

t con-  
maxi-

es are  
ly for-  
ability  
ys for-

et by



# 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/(29,447,726 - 796,084)) 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.

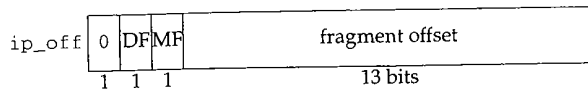


Figure 10.1 ip\_off controls fragmentation of an IP packet.

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.

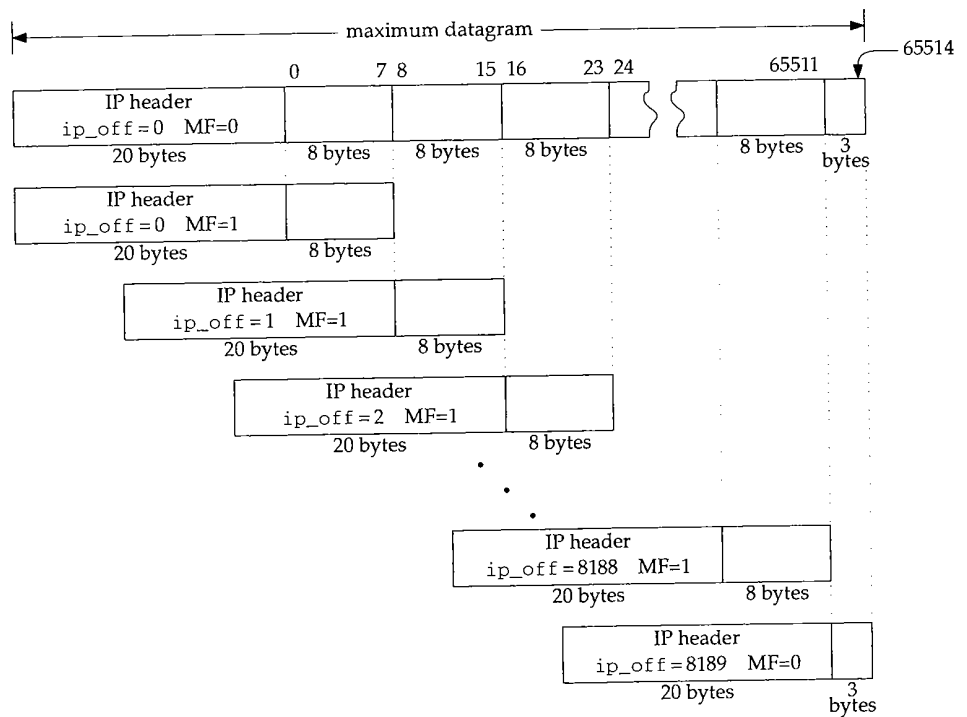


Figure 10.2 Fragmentation of a 65535-byte datagram.

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.

## 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.

File	Description
netinet/ip_var.h	reassembly data structures
netinet/ip_output.c	fragmentation code
netinet/ip_input.c	reassembly code

Figure 10.3 Files discussed in this chapter.

### Global Variables

Only one global variable, *ipq*, is described in this chapter.

Variable	Type	Description
<i>ipq</i>	struct ipq *	reassembly list

Figure 10.4 Global variable introduced in this chapter.

## 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.

ipstat member	Description
<code>ips_cantfrag</code>	#datagrams not sent because fragmentation was required but was prohibited by the DF bit
<code>ips_odropped</code>	#output packets dropped because of a memory shortage
<code>ips_ofragments</code>	#fragments transmitted
<code>ips_fragmented</code>	#packets fragmented for output

Figure 10.5 Statistics collected in this chapter.

## 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),
- construct fragment list (Figure 10.7), and
- construct initial fragment and send fragments (Figure 10.8).

```

-----ip_output.c
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  }
-----ip_output.c

```

Figure 10.6 `ip_output` function: determine fragment size.

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 (`& ~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`.

Fig-  
ibedFig-  
trans-  
l and  
itself  
tation

output.c

output.c

ompli-  
tion is

```

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++;
315         }

```

Figure 10.7 ip\_output function: construct fragment list.

\_output.c

- 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`.

hain.

In Figure 10.8, `ip_output` constructs the initial fragment and then passes each fragment to the interface layer.

() {

```

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

Figure 10.8 `ip_output` function: send fragments.

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.

p\_output.c

## 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.

```

395 int
396 ip_optcopy(ip, jp)
397 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*

Figure 10.9 `ip_optcopy` function.

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`.

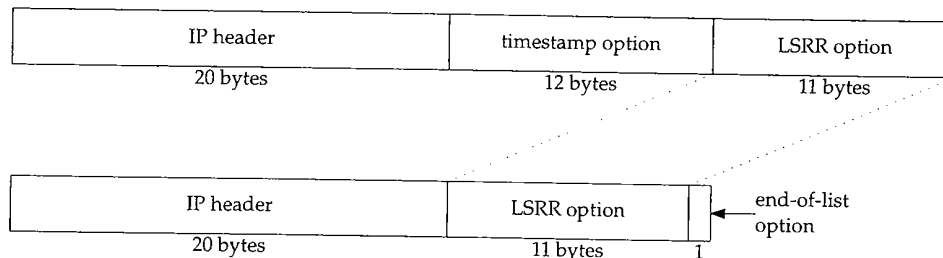


Figure 10.10 Not all options are copied during fragmentation.

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.

## 10.5 Reassembly

Now that we have described the fragmentation of a datagram (or of a fragment), we return to `ipintr` and the reassembly process. In Figure 8.15 we omitted the reassembly code from `ipintr` and postponed its discussion. `ipintr` can pass only entire datagrams up to the transport layer for processing. Fragments that are received by `ipintr` are passed to `ip_reass`, which attempts to reassemble fragments into complete datagrams. The code from `ipintr` is shown in Figure 10.11.

271-279 Recall that `ip_off` contains the DF bit, the MF bit, and the fragment offset. The DF bit is masked out and if either the MF bit or fragment offset is nonzero, the packet is a fragment that must be reassembled. If both are zero, the packet is a complete datagram, the reassembly code is skipped and the `else` clause at the end of Figure 10.11 is executed, which excludes the header length from the total datagram length.

280-286 `m_pullup` moves data in an external cluster into the data area of the `mbuf`. Recall that the SLIP interface (Section 5.3) may return an entire IP packet in an external cluster if it does not fit in a single `mbuf`. Also `m_devget` can return the entire packet in a cluster (Section 2.6). Before the `dtom` macro will work (Section 2.6), `m_pullup` must move the IP header from the cluster into the data area of an `mbuf`.

*ip\_input.c*

```

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_freem(fp);

```

input.c

```

323     } else
324         ip->ip_len -= hlen;

```

ip\_input.c

Figure 10.11 ipintr function: fragment processing.

287-297 Net/3 keeps incomplete datagrams on the global doubly linked list, `ipq`. The name is somewhat confusing since the data structure isn't a queue. That is, insertions and deletions can occur anywhere in the list, not just at the ends. We'll use the term *list* to emphasize this fact.

`ipintr` performs a linear search of the list to locate the appropriate datagram for the current fragment. Remember that fragments are uniquely identified by the 4-tuple: `{ip_id, ip_src, ip_dst, ip_p}`. Each entry in `ipq` is a list of fragments and `fp` points to the appropriate list if `ipintr` finds a match.

Net/3 uses linear searches to access many of its data structures. While simple, this method can become a bottleneck in hosts supporting large numbers of network connections.

298-303 At found, the packet is modified by `ipintr` to facilitate reassembly:

- 304
- `ipintr` changes `ip_len` to exclude the standard IP header and any options. We must keep this in mind to avoid confusion with the standard interpretation of `ip_len`, which includes the standard header, options, and data. `ip_len` is also changed if the reassembly code is skipped because this is not a fragment.
- 305-307
- `ipintr` copies the MF flag into the low-order bit of `ipf_mff`, which overlays `ip_tos` (`&= ~1` clears the low-order bit only). Notice that `ip` must be cast to a pointer to an `ipasfrag` structure before `ipf_mff` is a valid member. Section 10.6 and Figure 10.14 describe the `ipasfrag` structure.

Although RFC 1122 requires the IP layer to provide a mechanism that enables the transport layer to set `ip_tos` for every outgoing datagram, it only recommends that the IP layer pass `ip_tos` values to the transport layer at the destination host. Since the low-order bit of the TOS field must always be 0, it is available to hold the MF bit while `ip_off` (where the MF bit is normally found) is used by the reassembly algorithm.

`ip_off` can now be accessed as a 16-bit offset instead of 3 flag bits and a 13-bit offset.

- 308
- `ip_off` is multiplied by 8 to convert from 8-byte to 1-byte units.

`ipf_mff` and `ip_off` determine if `ipintr` should attempt reassembly. Figure 10.12 describes the different cases and the corresponding actions. Remember that `fp` points to the list of fragments the system has previously received for the datagram. Most of the work is done by `ip_reass`.

309-322 If `ip_reass` is able to assemble a complete datagram by combining the current fragment with previously received fragments, it returns a pointer to the reassembled datagram. If reassembly is not possible, `ip_reass` saves the fragment and `ipintr` jumps to next to process the next packet (Figure 8.12).

323-324 This `else` branch is taken when a complete datagram arrives and `ip_hlen` is modified as described earlier. This is the normal flow, since most received datagrams are not fragments.

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

Figure 10.12 IP fragment processing in `ipintr` and `ip_reass`.

If a complete datagram is available after reassembly processing, it is passed up to the appropriate transport protocol by `ipintr` (Figure 8.15):

```
(*inetsw[ip_protox[ip->ip_p]].pr_input)(m, hlen);
```

## 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, shown in Figure 10.13.

```

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*

*ip\_var.h*

Figure 10.13 `ipq` structure.

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.



```

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_MF) */
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*

*ip\_var.h*

Figure 10.14 ipasfrag structure.

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`).

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.

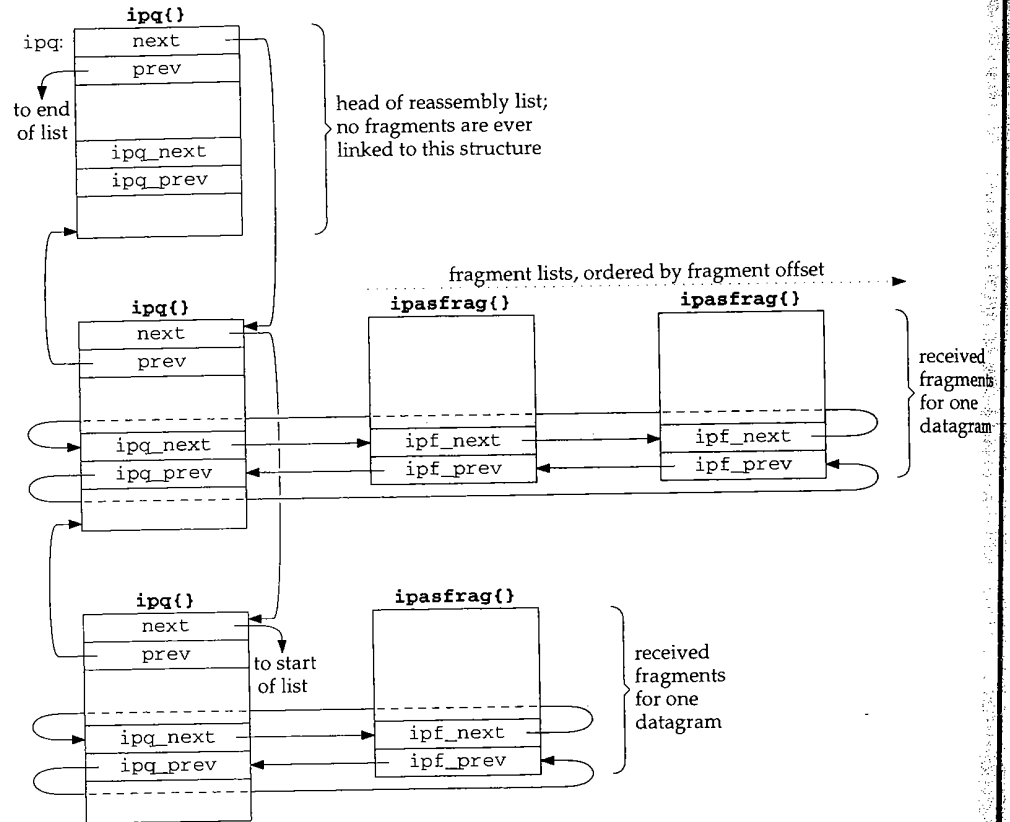


Figure 10.15 The fragment header list, `ipq`, and fragments.

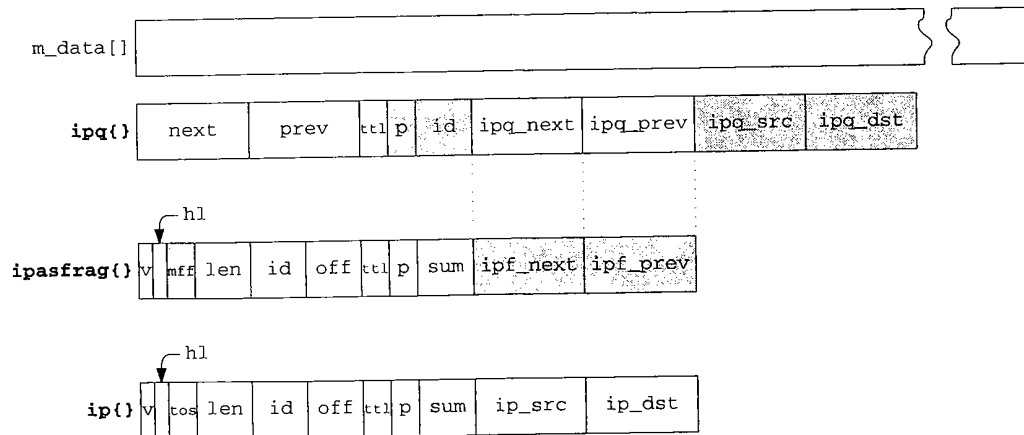


Figure 10.16 An area of memory can be accessed through multiple structures.

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.

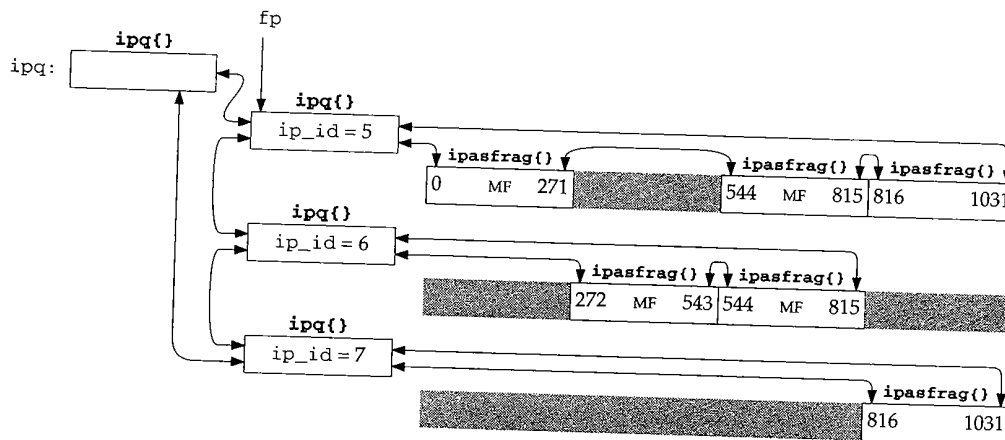


Figure 10.17 Reassembly of three IP datagrams.

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 contains 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.

```

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;
359
360     /* reassembly code */
361
362     dropfrag:
363     ipstat.ips_fragdropped++;
364     m_freem(m);
365     return (0);
366 }

```

*ip\_input.c*

Figure 10.18 `ip_reass` function: datagram reassembly.

343-358 When `ip_reass` is called, `ip` points to the fragment and `fp` either points to the matching `ipq` structure or is null.

under a  
e data-  
r frag-  
header,  
ler and  
of data

tes 272  
the end  
gments,

in an IP

p\_input.c

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.

```

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 = IPPFRAGTTL;
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

Figure 10.19 ip\_reass function: create reassembly list.

### 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.

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.

-ip\_input.c

nts to the

Function	Description
insque	Insert <i>node</i> just after <i>prev</i> . void <b>insque</b> (void * <i>node</i> , void * <i>prev</i> );
remque	Remove <i>node</i> from list. void <b>remque</b> (void * <i>node</i> );
ip_enq	Insert fragment <i>p</i> just after fragment <i>prev</i> . void <b>ip_enq</b> (struct ipasfrag * <i>p</i> , struct ipasfrag * <i>prev</i> );
ip_deq	Remove fragment <i>p</i> . void <b>ip_deq</b> (struct ipasfrag * <i>p</i> );

Figure 10.20 Queueing functions used by ip\_reass.

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

367

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.

```

376      /*
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;

```

*ip\_input.c*

*ip\_input.c*

Figure 10.21 `ip_reass` function: find position in reassembly list.

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.

In the following discussion, an *earlier* fragment is a fragment that previously arrived at the host.

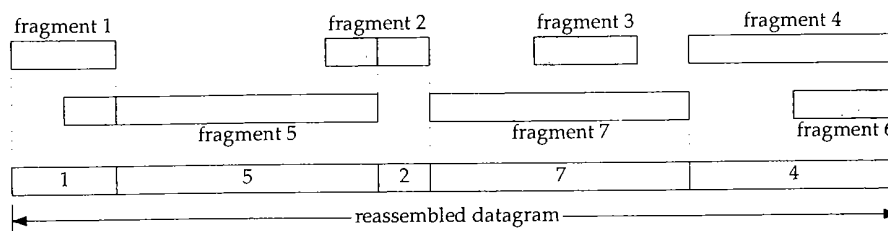


Figure 10.22 The byte range of fragments may overlap at the destination.

The code in Figure 10.23 trims or discards incoming fragments.

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.

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.

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.

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.

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.



```

382  /*
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  }

```

Figure 10.23 ip\_reass function: trim incoming packet.

```

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  }

```

Figure 10.24 ip\_reass function: trim existing packets.

```

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);

```

Figure 10.25 ip\_reass function: insert packet.

```

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

Figure 10.26 ip\_reass function: reassemble datagram.

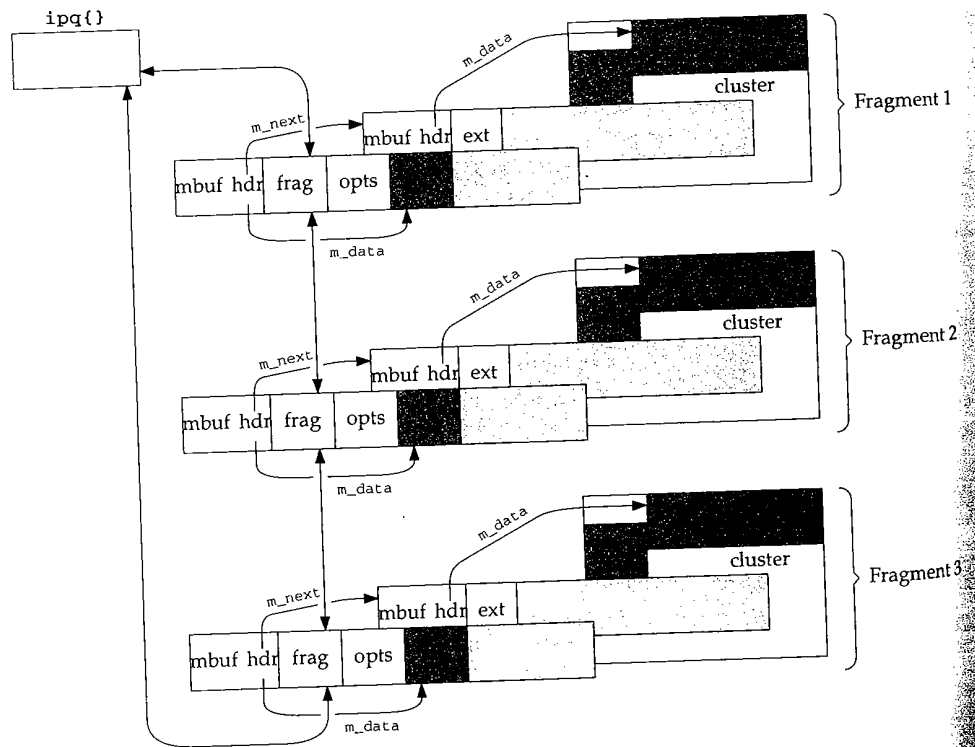


Figure 10.27 m\_cat reassembles the fragments within mbufs.

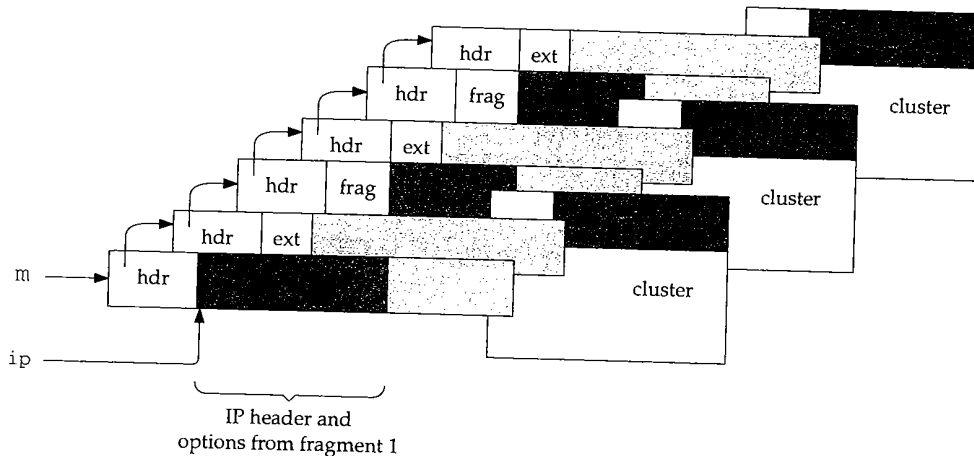


Figure 10.28 The reassembled datagram.

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.

```

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->m_next)
461          plen += m->m_len;
462      t->m_pkthdr.len = plen;
463  }
464  return ((struct ip *) ip);

```

Figure 10.29 `ip_reass` function: datagram reassembly.

**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.

**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.

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.

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.

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.

```

515 void
516 ip_slowtimo(void)
517 {
518     struct ipq *fp;
519     int     s = splnet();
520
521     fp = ipq.next;
522     if (fp == 0) {
523         splx(s);
524         return;
525     }
526     while (fp != &ipq) {
527         --fp->ipq_ttl;
528         fp = fp->next;
529         if (fp->prev->ipq_ttl == 0) {
530             ipstat.ips_fragtimeout++;
531             ip_freef(fp->prev);
532         }
533     }
534     splx(s);
535 }

```

Figure 10.30 ip\_slowtimo function.

```

474 void
475 ip_freef(fp)
476 struct ipq *fp;
477 {
478     struct ipasfrag *q, *p;
479
480     for (q = fp->ipq_next; q != (struct ipasfrag *) fp; q = p) {
481         p = q->ipf_next;
482         ip_deq(q);
483         m_freem(dtom(q));
484     }
485     remque(fp);
486     (void) m_free(dtom(fp));
487 }

```

Figure 10.31 ip\_freef function.

```

538 void
539 ip_drain()
540 {
541     while (ipq.next != &ipq) {
542         ipstat.ips_fragdropped++;
543         ip_freef(ipq.next);
544     }
545 }

```

Figure 10.32 ip\_drain function.

## 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.
- 10.4 [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. 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.

- 10.5 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?

# ***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.

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
<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_HOST PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_HOST PRC_UNREACH_HOST
<i>ICMP_TIMXCEED</i> <i>ICMP_TIMXCEED_INTRANS</i> <i>ICMP_TIMXCEED_REASS</i> other	time exceeded IP time-to-live expired in transit reassembly 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	

Figure 11.1 ICMP message types and codes.



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	
other	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
13	rip_input			
14	rip_input			
15	rip_input			
other	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	
other	rip_input			
<i>ICMP_PARAMPROB</i>				
0	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
<i>ICMP_PARAMPROB_OPTABSENT</i>	pr_ctlinput	udp_notify	tcp_notify	ENOPROTOOPT
other	rip_input			
<i>ICMP_SOURCEQUENCH</i>	pr_ctlinput	udp_notify	tcp_quench	
other	rip_input			

Figure 11.2 ICMP message types and codes (continued).

Figures 11.1 and 11.2 contain a lot of information:

- 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.

Function	Description	Reference
<code>icmp_reflect</code>	generate reply to ICMP request	Section 11.12
<code>in_rtchange</code>	update IP routing tables	Figure 22.34
<code>pfctlinput</code>	report error to all protocols	Section 7.7
<code>pr_ctlinput</code>	report error to the protocol associated with the socket	Section 7.4
<code>rip_input</code>	process unrecognized ICMP messages	Section 32.5
<code>tcp_notify</code>	ignore or report error to process	Figure 27.12
<code>tcp_quench</code>	slow down the output	Figure 27.13
<code>udp_notify</code>	report error to process	Figure 23.31

Figure 11.3 Functions called during ICMP input processing.

- 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 proto-

col 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.

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

Figure 11.4 ICMP message processing.

### 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.

File	Description
netinet/ip_icmp.h	ICMP structure definitions
netinet/ip_icmp.c	ICMP processing

Figure 11.5 Files discussed in this chapter.

#### Global Variables

The global variables shown in Figure 11.6 are 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)

Figure 11.6 Global variables introduced in this chapter.

## Statistics

Statistics are collected by the members of the `icmpstat` structure shown in Figure 11.7.

icmpstat member	Description	Used by SNMP
<code>icps_oldicmp</code>	#errors discarded because datagram was an ICMP message	•
<code>icps_oldshort</code>	#errors discarded because IP datagram was too short	•
<code>icps_badcode</code>	#ICMP messages discarded because of an invalid code	•
<code>icps_badlen</code>	#ICMP messages discarded because of an invalid ICMP body	•
<code>icps_checksum</code>	#ICMP messages discarded because of a bad ICMP checksum	•
<code>icps_tooshort</code>	#ICMP messages discarded because of a short ICMP header	•
<code>icps_outhist[]</code>	array of output counters; one for each ICMP type	•
<code>icps_inhist[]</code>	array of input counters; one for each ICMP type	•
<code>icps_error</code>	#of calls to <code>icmp_error</code> (excluding redirects)	
<code>icps_reflect</code>	#ICMP messages reflected by the kernel	

Figure 11.7 Statistics collected in this chapter.

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.

netstat -s output	icmpstat member
84124 calls to <code>icmp_error</code>	<code>icps_error</code>
0 errors not generated 'cuz old message was icmp	<code>icps_oldicmp</code>
Output histogram:	<code>icps_outhist[]</code>
echo reply: 11770	<code>ICMP_ECHOREPLY</code>
destination unreachable: 84118	<code>ICMP_UNREACH</code>
time exceeded: 6	<code>ICMP_TIMXCEED</code>
6 messages with bad code fields	<code>icps_badcode</code>
0 messages < minimum length	<code>icps_badlen</code>
0 bad checksums	<code>icps_checksum</code>
143 messages with bad length	<code>icps_tooshort</code>
Input histogram:	<code>icps_inhist[]</code>
echo reply: 793	<code>ICMP_ECHOREPLY</code>
destination unreachable: 305869	<code>ICMP_UNREACH</code>
source quench: 621	<code>ICMP_SOURCEQUENCH</code>
routing redirect: 103	<code>ICMP_REDIRECT</code>
echo: 11770	<code>ICMP_ECHO</code>
time exceeded: 25296	<code>ICMP_TIMXCEED</code>
11770 message responses generated	<code>icps_reflect</code>

Figure 11.8 Sample ICMP statistics.

## SNMP Variables

Figure 11.9 shows the relationship between the variables in the SNMP ICMP group and the statistics collected by Net/3.

11.7.

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 icmpInSrcQuenchs 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 icmpOutSrcQuenchs icmpOutRedirects icmpOutEchos icmpOutEchoReps icmpOutTimestamps icmpOutTimestampReps icmpOutAddrMasks icmpOutAddrMaskReps	icps_outhist[] counter	#ICMP messages sent for each type

Figure 11.9 Simple SNMP variables in ICMP group.

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.

and

### 11.3 icmp Structure

Net/3 accesses an ICMP message through the `icmp` structure shown in Figure 11.10.

```

-----ip_icmp.h
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

```

Figure 11.10 icmp structure.

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.

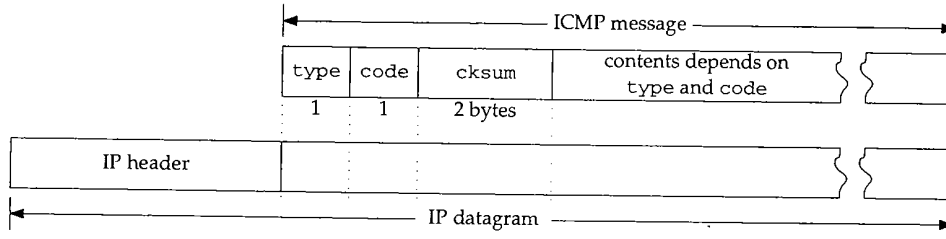


Figure 11.11 An ICMP message (`icmp_omitted`).

### 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.

Member	inetsw[4]	Description
<code>pr_type</code>	<code>SOCK_RAW</code>	ICMP provides raw packet services
<code>pr_domain</code>	<code>&amp;inetdomain</code>	ICMP is part of the Internet domain
<code>pr_protocol</code>	<code>IPPROTO_ICMP (1)</code>	appears in the <code>ip_p</code> field of the IP header
<code>pr_flags</code>	<code>PR_ATOMIC/PR_ADDR</code>	socket layer flags, not used by ICMP
<code>pr_input</code>	<code>icmp_input</code>	receives ICMP messages from the IP layer
<code>pr_output</code>	<code>rip_output</code>	sends ICMP messages to the IP layer
<code>pr_ctlinput</code>	<code>0</code>	not used by ICMP
<code>pr_ctloutput</code>	<code>rip_ctloutput</code>	respond to administrative requests from a process
<code>pr_usrreq</code>	<code>rip_usrreq</code>	respond to communication requests from a process
<code>pr_init</code>	<code>0</code>	not used by ICMP
<code>pr_fasttimo</code>	<code>0</code>	not used by ICMP
<code>pr_slowtimo</code>	<code>0</code>	not used by ICMP
<code>pr_drain</code>	<code>0</code>	not used by ICMP
<code>pr_sysctl</code>	<code>icmp_sysctl</code>	modify ICMP parameters

Figure 11.12 ICMP `inetsw` entry.

## 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]`.

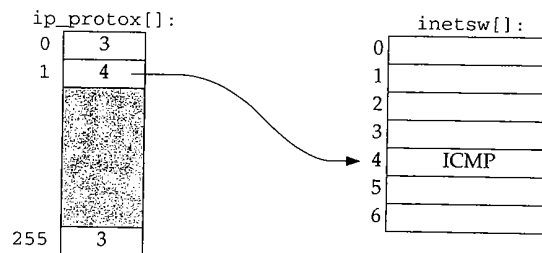


Figure 11.13 An `ip_p` value of 1 selects `inetsw[4]`.

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.

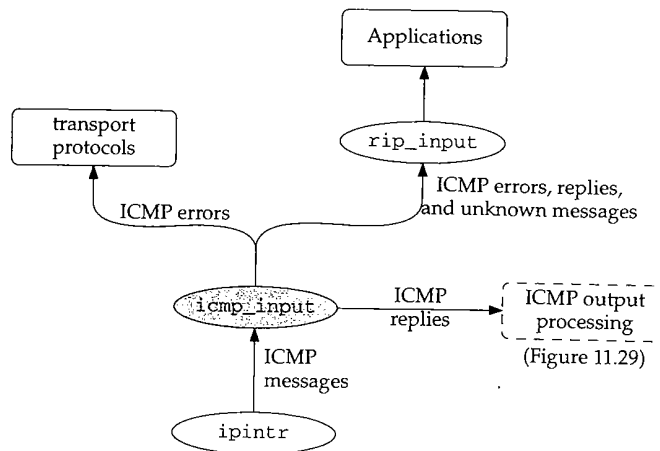


Figure 11.14 ICMP input processing.

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.



```

131 static struct sockaddr_in icmpsrc = { 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_protoc[];

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

Figure 11.15 icmp\_input function.

### 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 `icmptype` 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 `icmptype` 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, `icmptype` 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`.

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>	$36 + optsize$	size of an ICMP error message including <i>optsize</i> bytes of IP options from the invalid packet <i>p</i> .

Figure 11.16 Constants and a macro referenced by ICMP to validate messages.

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 `tracert`).
- 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.

**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.

-ip\_icmp.c

ic alloca-  
ck since  
o\_inputtemporary  
version of  
end of the  
input func-  
instead ofMP mes-  
1.16 lists  
ssages inres it in  
e header  
oshort  
ader are  
id the IP

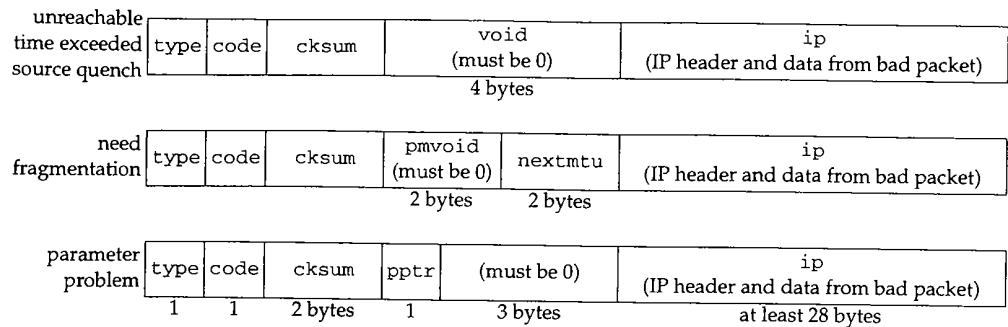


Figure 11.17 ICMP error messages (icmp\_ omitted).

The code in Figure 11.18 is from the switch shown in Figure 11.15.

```

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;

```

*ip\_icmp.c*

```

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

Figure 11.18 icmp\_input function: error messages.

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 prevents 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 (`icmpsrc`), 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.

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

Figure 11.19 The protocol-independent error codes.

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.

## 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.

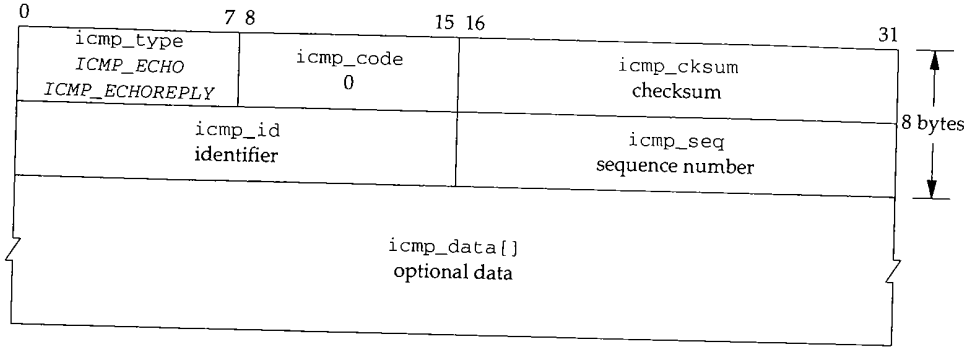


Figure 11.20 ICMP echo request and reply.

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.

```

235     case ICMP_ECHO:
236         icp->icmp_type = ICMP_ECHOREPLY;
237         goto reflect;

                /* 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;
    
```

Figure 11.21 icmp\_input function: echo request and reply.

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.

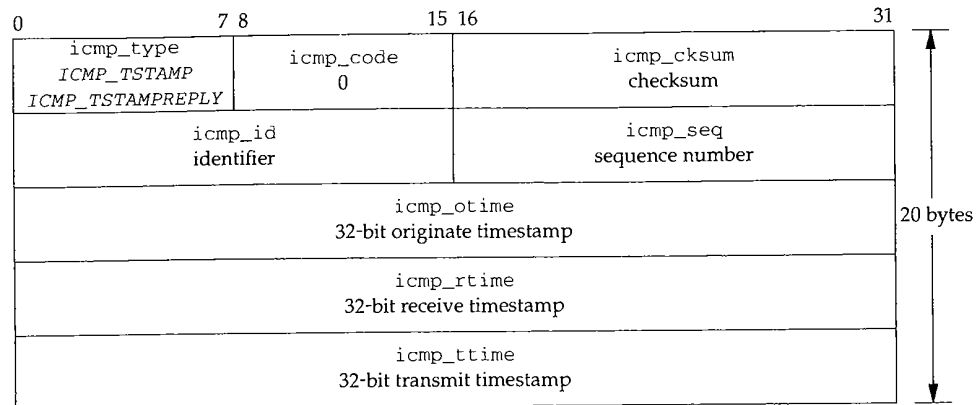


Figure 11.22 ICMP timestamp request and reply.

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.

```

238     case ICMP_TSTAMP:
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;

```

*ip\_icmp.c*

Figure 11.23 icmp\_input function: timestamp request and reply.

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.

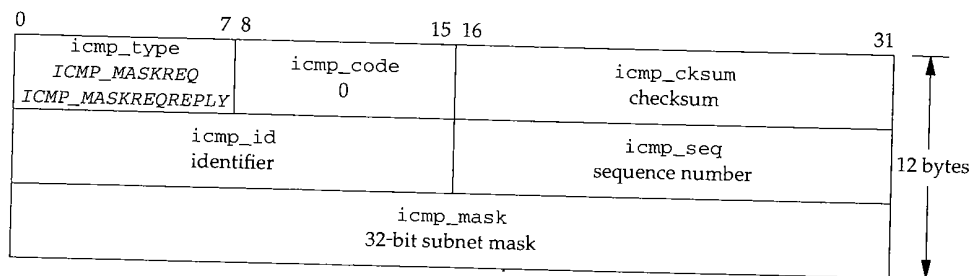


Figure 11.24 ICMP address mask request and reply.

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.

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

```

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
259     case INADDR_BROADCAST:
260     case INADDR_ANY:
261         icmpdst.sin_addr = ip->ip_src;
262         break;
263
264     default:
265         icmpdst.sin_addr = ip->ip_dst;
266     }
267     ia = (struct in_ifaddr *) ifaof_ifpforaddr(
268         (struct sockaddr *) &icmpdst, m->m_pkthdr.rcvif);
269     if (ia == 0)
270         break;
271     icp->icmp_type = ICMP_MASKREPLY;
272     icp->icmp_mask = ia->ia_sockmask.sin_addr.s_addr;
273     if (ip->ip_src.s_addr == 0) {
274         if (ia->ia_ifp->if_flags & IFF_BROADCAST)
275             ip->ip_src = satosin(&ia->ia_broadaddr)->sin_addr;
276         else if (ia->ia_ifp->if_flags & IFF_POINTOPOINT)
277             ip->ip_src = satosin(&ia->ia_dstaddr)->sin_addr;
278     }

```

Figure 11.25 icmp\_input function: address mask request and reply.

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.

## 11.8 Redirect Processing

Figure 11.26 shows the format of ICMP redirect messages.

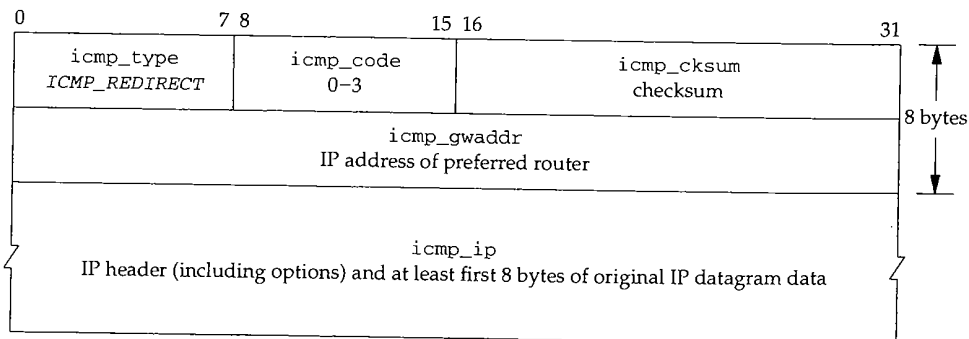


Figure 11.26 ICMP redirect message.

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.

```

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         icmpsrc.sin_addr = icp->icmp_ip.ip_dst;
301         rtredirect((struct sockaddr *) &icmpsrc,
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 *) &icmpsrc);
306         break;

```

*ip\_icmp.c*

Figure 11.27 `icmp_input` function: redirect messages.

### 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 `icmpsrc`, 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, `icmpsrc` 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 `icmpsrc`) 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.

**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`.

icmp.c

```

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;

```

message  
has an  
11.16  
message,  
ing any  
c, the  
ecom-  
estina-

Figure 11.28 `icmp_input` function: reply messages.

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`.

ing the

## 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.

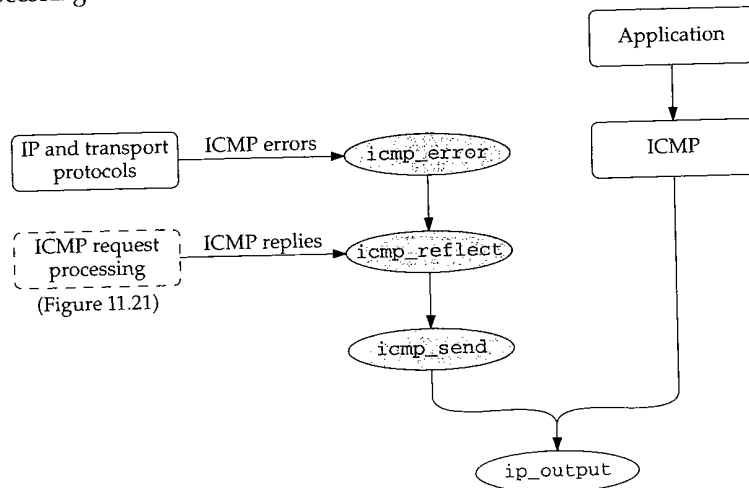


Figure 11.29 ICMP output processing.

## 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).

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. `mto` 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.

3 that IP  
messages  
messages  
to ICMP

```

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 *icp;
56     struct mbuf *m;
57     unsigned icmplen;
58
59     if (type != ICMP_REDIRECT)
60         icmpstat.icps_error++;
61     /*
62      * Don't send error if not the first fragment of message.
63      * Don't error if the old packet protocol was ICMP
64      * error message, only known informational types.
65      */
66     if (oip->ip_off & ~(IP_MF | IP_DF))
67         goto freeit;
68     if (oip->ip_p == IPPROTO_ICMP && type != ICMP_REDIRECT &&
69         n->m_len >= oiplen + ICMP_MINLEN &&
70         !ICMP_INFOTYPE(((struct icmp *) ((caddr_t) oip + oiplen))->icmp_type)) {
71         icmpstat.icps_oldicmp++;
72         goto freeit;
73     }
74     /* Don't send error in response to a multicast or broadcast packet */
75     if (n->m_flags & (M_BCAST | M_MCAST))
76         goto freeit;

```

Figure 11.30 icmp\_error function: validation.

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).

IP or the  
e source

atagram;  
p router  
outgoing  
a pointer  
s kept in

t/3 does  
ible.

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.

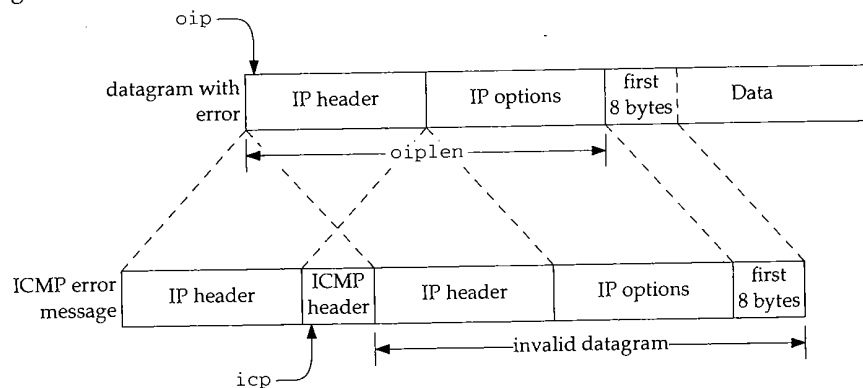


Figure 11.31 The construction of an ICMP error message.

The code in Figure 11.32 builds the error message.

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.



```

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;

```

Figure 11.32 icmp\_error function: message header construction.

- 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.

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.

```

107     bcopy((caddr_t) oip, (caddr_t) & icp->icmp_ip, icmplen);
108     nip = &icp->icmp_ip;
109     nip->ip_len = htons((u_short) (nip->ip_len + oiplen));

110     /*
111     * Now, copy old ip header (without options)
112     * in front of icmp message.
113     */
114     if (m->m_data - sizeof(struct ip) < m->m_pktdat)
115         panic("icmp len");
116     m->m_data -= sizeof(struct ip);
117     m->m_len += sizeof(struct ip);
118     m->m_pkthdr.len = m->m_len;
119     m->m_pkthdr.rcvif = n->m_pkthdr.rcvif;
120     nip = mtod(m, struct ip *);
121     bcopy((caddr_t) oip, (caddr_t) nip, sizeof(struct ip));
122     nip->ip_len = m->m_len;
123     nip->ip_hl = sizeof(struct ip) >> 2;
124     nip->ip_p = IPPROTO_ICMP;
125     nip->ip_tos = 0;
126     icmp_reflect(m);

127     freeit:
128     m_freem(n);
129 }

```

Figure 11.33 icmp\_error function: including the original datagram.

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.

<sup>126-129</sup> The completed message is passed to `icmp_reflect`, where it is sent back to the source host. The invalid datagram is discarded.

## 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.

mp.c

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)

Figure 11.34 ICMP discard and address summary.

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.

**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.

icmp.c

ld be  
ns) is

inction  
e stan-

len),

to the

r back  
lect  
.. The  
.. Fig-

```

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_NS_SHIFT))) {
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;

```

Figure 11.35 icmp\_reflect function: address selection.

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

mp.c

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.

```

372     if (optlen > 0) {
373         u_char *cp;
374         int     opt, cnt;
375         u_int   len;

376         /*
377          * Retrieve any source routing from the incoming packet;
378          * add on any record-route or timestamp options.
379          */
380         cp = (u_char *) (ip + 1);
381         if ((opts = ip_srcroute()) == 0 &&
382             (opts = m_gethdr(M_DONTWAIT, MT_HEADER))) {
383             opts->m_len = sizeof(struct in_addr);
384             mtod(opts, struct in_addr *)->s_addr = 0;
385         }
386         if (opts) {
387             for (cnt = optlen; cnt > 0; cnt -= len, cp += len) {
388                 opt = cp[IPOPT_OPTVAL];
389                 if (opt == IPOPT_EOL)
390                     break;
391                 if (opt == IPOPT_NOP)
392                     len = 1;
393                 else {
394                     len = cp[IPOPT_OLLEN];
395                     if (len <= 0 || len > cnt)
396                         break;
397                 }
398                 /*
399                  * Should check for overflow, but it "can't happen"
400                  */
401                 if (opt == IPOPT_RR || opt == IPOPT_TS ||
402                     opt == IPOPT_SECURITY) {
403                     bcopy((caddr_t) cp,
404                         mtod(opts, caddr_t) + opts->m_len, len);
405                     opts->m_len += len;
406                 }
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         }

```

*ip\_icmp.c*

Figure 11.36 `icmp_reflect` function: option construction.

te and  
hed to  
n how  
s these

**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`.

`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.

```

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 }

```

ip\_icmp.c

Figure 11.37 `icmp_reflect` function: final assembly.

**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`.

**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.

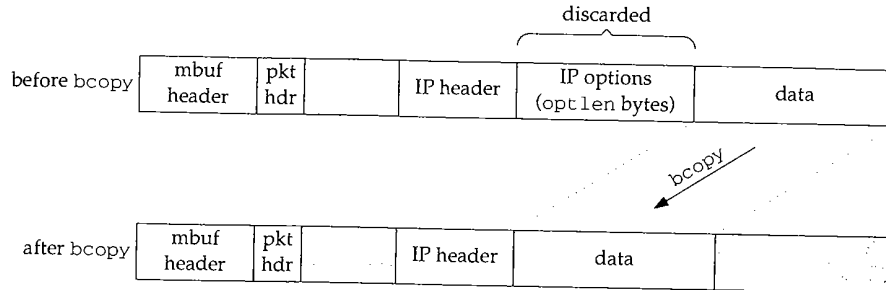


Figure 11.38 icmp\_reflect: removal of options.

### 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.

```

440 void                                     ip_icmp.c
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 }

```

Figure 11.39 icmp\_send function.

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).

### 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.

sysctl constant	Net/3 variable	Description
ICMPCTL_MASKREPL	icmpmaskrepl	Should system respond to ICMP address mask requests?

Figure 11.40 `icmp_sysctl` parameters.

Figure 11.41 shows the `icmp_sysctl` function.

```

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 }

```

*ip\_icmp.c*

*ip\_icmp.c*

Figure 11.41 `icmp_sysctl` function.

- 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.



## 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?
- 11.4 If the ICMP information request is obsolete, why does `icmp_input` pass it to `rip_input` instead of discarding it?
- 11.5 We pointed out that Net/3 does not convert the offset and length field of an IP packet to network byte order before including the packet in an ICMP error message. Why is this inconsequential in the case of the IP offset field?
- 11.6 Describe a situation in which `ifaof_ifpforaddr` from Figure 11.25 returns a null pointer.
- 11.7 What happens to data included after the timestamps in a timestamp query?
- 11.8 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.

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 timestamp 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.
- 11.10 In Figure 11.30, what happens to a packet that has the high-order bit of `ip_off` set?
- 11.11 Why is the return value from `ip_output` discarded in Figure 11.39?



## 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.

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	

Figure 12.1 Some registered IP multicast groups.

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 `mroute`, 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.

## 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.

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_etherubr.c	Ethernet multicast functions
netinet/in.c	group membership functions
netinet/ip_input.c	input multicast processing
netinet/ip_output.c	output multicast processing

Figure 12.2 Files discussed in this chapter.

### Global Variables

Three new global variables are 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_mrouter	struct socket *	pointer to socket created by multicast routing daemon

Figure 12.3 Global variables introduced in this chapter.

### Statistics

The code in this chapter updates a few of the counters maintained in the global `ipstat` structure.

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

Figure 12.4 Multicast processing statistics.

Link-level multicast statistics are collected in the `ifnet` structure (Figure 4.5) and may include multicasting of protocols other than IP.

### 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.

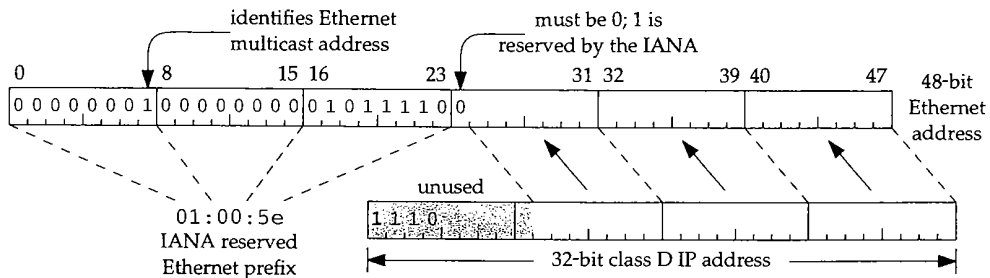


Figure 12.5 Mapping between IP and Ethernet addresses.

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

ved for IP  
ved for IP  
ng daemon

ipstat

4.5) and

Section 12.14 we'll see how this affects input processing. Figure 12.6 shows the macro that implements this mapping in Net/3.

```

61 #define ETHER_MAP_IP_MULTICAST(ipaddr, enaddr) \
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*

Figure 12.6 ETHER\_MAP\_IP\_MULTICAST macro.

#### 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.

## 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:

```

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 };

```

*if\_ether.h*

Figure 12.7 ether\_multi structure.

#### 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.

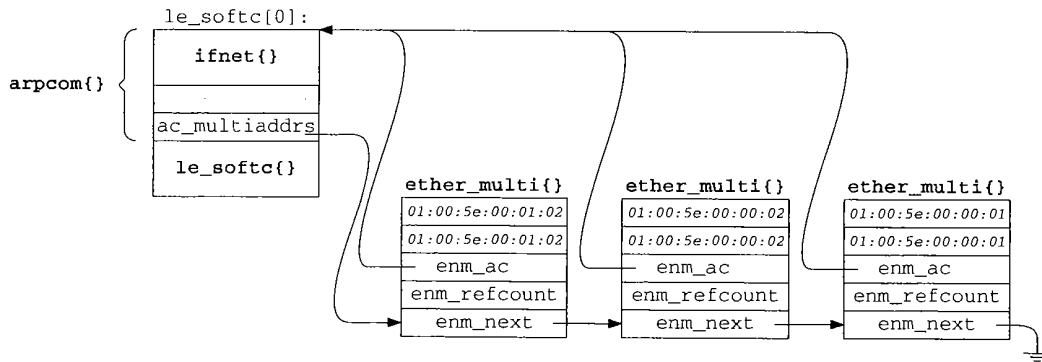


Figure 12.8 The LANCE interface with three ether\_multi structures.

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.

```

166 #define ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm) \
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         (bcmp((enm)->enm_addrlo, (addrlo), 6) != 0 || \
175          bcmp((enm)->enm_addrhi, (addrhi), 6) != 0); \
176         (enm) = (enm)->enm_next); \
177 }

```

Figure 12.9 ETHER\_LOOKUP\_MULTI macro.

### 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.

## 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.

Command	Argument	Function	Description
SIOCADDMULTI	struct ifreq *	ifioctl	add multicast address to reception list
SIOCDELMULTI	struct ifreq *	ifioctl	delete multicast address from reception list

Figure 12.10 Multicast `ioctl` commands.

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).

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.

her.h

```

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));

```

Figure 12.11 ifioctl function: multicast commands.

## 12.6 in\_multi Structure

ther.h

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.

Net/3  
useful  
ed to  
oct1

```

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 };

```

Figure 12.12 in\_multi structure.

device

### IP multicast addresses

t have  
passed

*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.

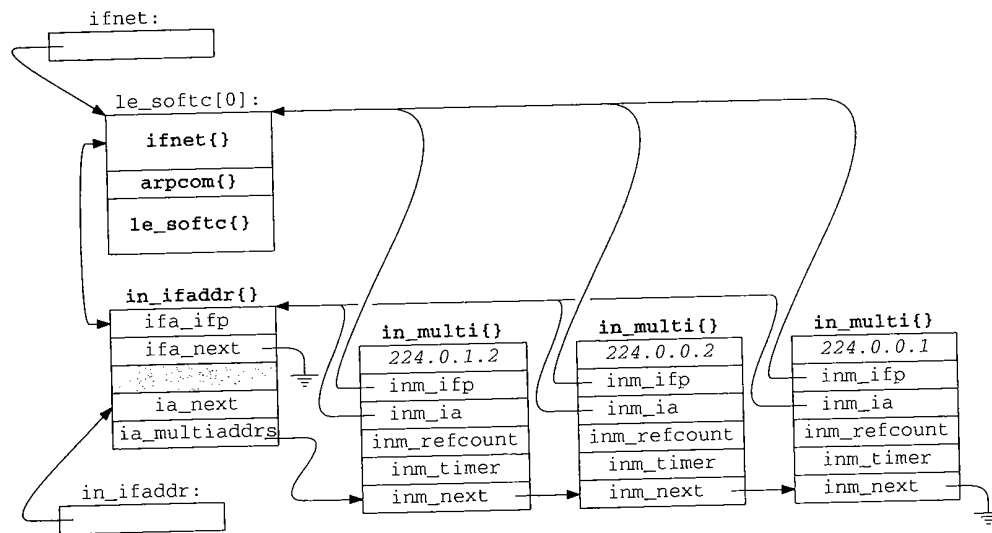


Figure 12.13 An IP multicast group list for the `le` interface.

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.

#### 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.

rtified  
roup}  
r pair,  
e pro-  
ased.  
refer-

ter 13.

ldress,

```

131 #define IN_LOOKUP_MULTI(addr, ifp, inm) \
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_multiaddrs; \
143             (inm) != NULL && (inm)->inm_addr.s_addr != (addr).s_addr; \
144             (inm) = inm->inm_next) \
145             continue; \
146 }

```

Figure 12.14 IN\_LOOKUP\_MULTI macro.

## 12.7 ip\_moptions Structure

The `ip_moptions` structure contains the multicast options through which the transport layer controls multicast output processing. For example, the UDP call to `ip_output` is:

```

error = ip_output(m, inp->inp_options, &inp->inp_route,
                 inp->inp_socket->so_options & (SO_DONTROUTE|SO_BROADCAST),
                 inp->inp_moptions);

```

In Chapter 22 we'll see that `inp` points to an Internet protocol control block (PCB) and that UDP associates a PCB with each socket created by a process. Within the PCB, `inp_moptions` is a pointer to an `ip_moptions` structure. From this we see that a different `ip_moptions` structure may be passed to `ip_output` for each outgoing datagram. Figure 12.15 shows the definition of the `ip_moptions` structure.

```

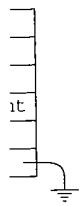
100 struct ip_moptions {
101     struct ifnet *imo_multicast_ifp; /* ifp for outgoing multicasts */
102     u_char imo_multicast_ttl; /* TTL for outgoing multicasts */
103     u_char imo_multicast_loop; /* 1 => hear sends if a member */
104     u_short imo_num_memberships; /* no. memberships this socket */
105     struct in_multi *imo_membership[IP_MAX_MEMBERSHIPS];
106 };

```

Figure 12.15 ip\_moptions structure.

### Multicast options

100-106 `ip_output` routes outgoing multicast datagrams through the interface pointed to by `imo_multicast_ifp` or, if `imo_multicast_ifp` is null, through the default interface for the destination multicast group (Chapter 14).



see Fig-  
anaged  
d to its  
' multi-

oup list  
ss list,  
ified by  
t. After

`imo_multicast_ttl` specifies the initial IP TTL value for outgoing multicasts. The default is 1, which causes multicast datagrams to remain on the local network.

If `imo_multicast_loop` is 0, the multicast datagram is not looped back and delivered to the transmitting interface even if the interface is a member of the multicast group. If `imo_multicast_loop` is 1, the multicast datagram is looped back to the transmitting interface if the interface is a member of the multicast group.

Finally, the integer `imo_num_memberships` and the array `imo_membership` maintain the list of {interface, group} pairs associated with the structure. Changes to the list are communicated to IP, which announces membership changes on the locally attached network. Each entry in the `imo_membership` array is a pointer to an `in_multi` structure attached to the `in_ifaddr` structure of the appropriate interface.

## 12.8 Multicast Socket Options

Several IP-level socket options, shown in Figure 12.10, provide process-level access to `ip_moptions` structures.

Command	Argument	Function	Description
<code>IP_MULTICAST_IF</code>	struct <code>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>	struct <code>ip_mreq</code>	<code>ip_ctloutput</code>	join a multicast group
<code>IP_DROP_MEMBERSHIP</code>	struct <code>ip_mreq</code>	<code>ip_ctloutput</code>	leave a multicast group

Figure 12.16 Multicast socket options.

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.

486-491 All the multicast options are handled through the `ip_setmoptions` and  
539-549 `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. `ENPROTOTOPT` is the more reasonable choice.

## 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

```

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     case PRCO_GETOPT:
502         switch (optname) {
503             /* other get cases */
504
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         }

```

Figure 12.17 ip\_ctloutput function: multicast options.

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.

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	

Figure 12.18 TTL values for IP multicast datagrams.

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.

## 12.10 ip\_setsockopt Function

The bulk of the `ip_setsockopt` function consists of a switch statement to handle each option. Figure 12.19 shows the beginning and end of `ip_setsockopt`. The body of the switch is discussed in the following sections.

<sup>650-664</sup> 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_setsockopt` modifies the structure it points to. Otherwise, `ip_setsockopt` allocates a new `ip_moptions` structure and saves its address in `*imop`. If no memory is available, `ip_setsockopt` 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

<sup>665-679</sup> When a new `ip_moptions` structure is allocated, `ip_setsockopt` 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

<sup>680-860</sup> The body of `ip_setsockopt` consists of a switch statement with a case for each option. The default case (for unknown options) sets `error` to `EOPNOTSUPP`.

```

650 int
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
682         /* switch cases */
683
684     default:
685         error = EOPNOTSUPP;
686         break;
687     }
688     /*
689     * If all options have default values, no need to keep the structure.
690     */
691     if (imo->imo_multicast_ifp == NULL &&
692         imo->imo_multicast_ttl == IP_DEFAULT_MULTICAST_TTL &&
693         imo->imo_multicast_loop == IP_DEFAULT_MULTICAST_LOOP &&
694         imo->imo_num_memberships == 0) {
695         free(*imop, M_IPMOPTS);
696         *imop = NULL;
697     }
698     return (error);
699 }

```

Figure 12.19 ip\_setmoptions function.

ut.c

**Discard structure if defaults are OK**

861-872 After the switch statement, `ip_setsockopt` examines the `ip_moptions` structure. If all the multicast options match their respective default values, the structure is unnecessary and is released. `ip_setsockopt` 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_setsockopt` 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.

```

681     case IP_MULTICAST_IF:                                     ip_output.c
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;

```

Figure 12.20 `ip_setsockopt` function: selecting a multicast output interface.

**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_setsockopt` 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

put.c

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.

```

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 = *(mto(m, u_char *));
720         break;

```

*ip\_output.c*

Figure 12.21 `ip_setmoptions` function: selecting an explicit multicast TTL.

#### 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.

```

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 = *(mtd(m, u_char *))) > 1) {
728             error = EINVAL;
729             break;
730         }
731         imo->imo_multicast_loop = loop;
732         break;

```

*ip\_output.c*

*ip\_output.c*

Figure 12.22 `ip_setsockopt` function: selecting multicast loopbacks.

#### 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`.

imo_multicast-		Recipients			
		Outgoing Interface?	Local Network?	Remote Networks?	Other Interfaces?
_loop	_ttl				
1	0	•			
1	1	•	•		
1	>1	•	•	•	see text

Figure 12.23 Loopback and TTL effects on multicast scope.

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).

## 12.11 Joining an IP Multicast Group

Other than the IP all-hosts group, which the kernel automatically joins (Figure 6.19), membership in a group is driven by explicit requests from processes on the system. The process of joining (or leaving) a multicast group is more involved than the other

multicast options. The `in_multi` list for an interface must be modified as well as any link-layer multicast structures such as the `ether_multi` list we described for Ethernet.

The data passed in the mbuf when `optname` is `IP_ADD_MEMBERSHIP` is an `ip_mreq` structure shown in 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 };

```

Figure 12.24 `ip_mreq` structure.

`imr_multiaddr` specifies the multicast group and `imr_interface` identifies the interface by its associated unicast IP address. The `ip_mreq` structure specifies the {interface, group} pair for membership changes.

Figure 12.25 illustrates the functions involved with joining and leaving a multicast group associated with our example Ethernet interface.

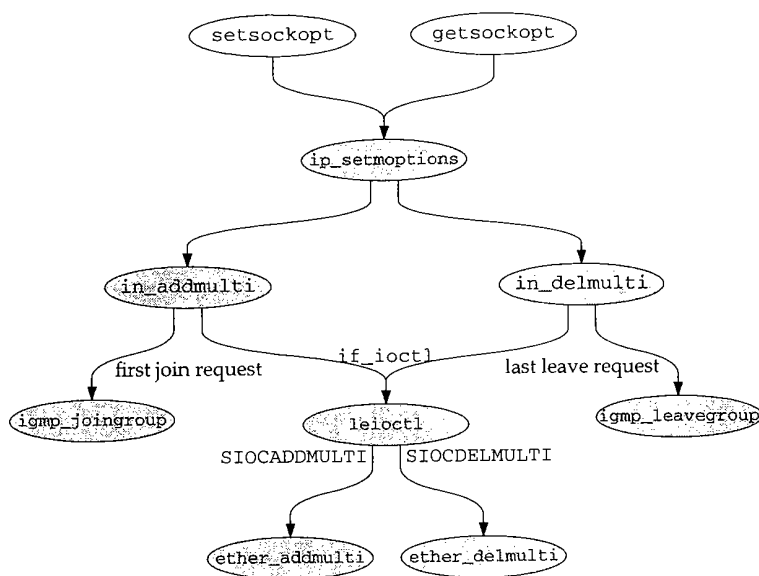


Figure 12.25 Joining and leaving a multicast group.

We start by describing the changes to the `ip_moptions` structure in the `IP_ADD_MEMBERSHIP` case in `ip_setmoptions` (Figure 12.26). Then we follow the request down through the IP layer, the Ethernet driver, and to the physical device—in our case, the LANCE Ethernet card.

```

733     case IP_ADD_MEMBERSHIP:                                     ip_output.c
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             rtalloc(&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;

```

*ip\_output.c*

Figure 12.26 `ip_setmoptions` function: joining a multicast group.

### Validation

733-746 `ip_setmoptions` starts by validating the request. If no mbuf was passed, if it is not the correct size, or if the address (`imr_multiaddr`) within the structure is not a multicast group, then `ip_setmoptions` posts `EINVAL`. `mreq` points to the valid `ip_mreq` structure.

### Locate the interface

747-774 If the unicast address of the interface (`imr_interface`) is `INADDR_ANY`, `ip_setmoptions` must locate the default interface for the specified group. A route structure is constructed with the group as the desired destination and passed to `rtalloc`, which locates a route for the group. If no route is available, the add request fails with the error `EADDRNOTAVAIL`. If a route is located, a pointer to the outgoing interface for the route is saved in `ifp` and the route entry, which is no longer needed, is released.

If `imr_interface` is not `INADDR_ANY`, an explicit interface has been requested. The macro `INADDR_TO_IFP` searches for the interface with the requested unicast address. If an interface isn't found or if it does not support multicasting, the request fails with the error `EADDRNOTAVAIL`.



We described the `route` structure in Section 8.5. The function `rtalloc` is described in Section 19.2, and the use of the routing tables for selecting multicast interfaces is described in Chapter 14.

### Already a member?

775-792 The last check performed on the request is to examine the `imo_membership` array to see if the selected interface is already a member of the requested group. If the `for` loop finds a match, or if the membership array is full, `EADDRINUSE` or `ETOOMANYREFS` is posted and processing of this option stops.

### Join the group

793-803 At this point the request looks reasonable. `in_addmulti` arranges for IP to begin receiving multicast datagrams for the group. The pointer returned by `in_addmulti` points to a new or existing `in_multi` structure (Figure 12.12) in the interface's multicast group list. It is saved in the membership array and the size of the array is incremented.

### `in_addmulti` Function

`in_addmulti` and its companion `in_delmulti` (Figures 12.27 and 12.36) maintain the list of multicast groups that an interface has joined. Join requests either add a new `in_multi` structure to the interface list or increase the reference count of an existing structure.

```

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*

Figure 12.27 `in_addmulti` function: first half.

### Already a member

469-487 `ip_setmoptions` has already verified that `ap` points to a class D multicast address and that `ifp` points to a multicast-capable interface. `IN_LOOKUP_MULTI` (Figure 12.14)

determines if the interface is already a member of the group. If it is a member, `in_addmulti` updates the reference count and returns.

If the interface is not yet a member of the group, the code in Figure 12.28 is executed.

```

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         IFP_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 }

```

Figure 12.28 `in_addmulti` function: second half.

member,

.28 is exe-

in.c

#### Update the `in_multi` list

487-509 If the interface isn't a member yet, `in_addmulti` allocates, initializes, and inserts the new `in_multi` structure at the front of the `ia_multiaddrs` list in the interface's `in_ifaddr` structure (Figure 12.13).

#### Update the interface and announce the change

510-530 If the interface driver has defined an `if_ioctl` function, `in_addmulti` constructs an `ifreq` structure (Figure 4.23) containing the group address and passes the `SIOCADDMULTI` request to the interface. If the interface rejects the request, the `in_multi` structure is unlinked from the interface and released. Finally, `in_addmulti` calls `igmp_joiningroup` to propagate the membership change to other hosts and routers.

`in_addmulti` returns a pointer to the `in_multi` structure or null if an error occurred.

#### `slioct1` and `loioctl` Functions: `SIOCADDMULTI` and `SIOCDELMULTI`

Multicast group processing for the SLIP and loopback interfaces is trivial: there is nothing to do other than error checking. Figure 12.29 shows the SLIP processing.

```

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;

```

Figure 12.29 `slioct1` function: multicast processing.

673-687 `EAFNOSUPPORT` is returned whether the request is empty or not for the `AF_INET` protocol family.

Figure 12.30 shows the loopback processing.

152-166 The processing for the loopback interface is identical to the SLIP code in Figure 12.29. `EAFNOSUPPORT` is returned whether the request is empty or not for the `AF_INET` protocol family.

in.c

```

-----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
161         case AF_INET:
162             break;
163
164         default:
165             error = EAFNOSUPPORT;
166             break;
167         }
168         break;
-----if_loop.c

```

Figure 12.30 `leioctl` function: multicast processing.**leioctl Function: SIOCADDMULTI and SIOCDELMULTI**

Recall from Figure 4.2 that `leioctl` is the `if_ioctl` function for the LANCE Ethernet driver. Figure 12.31 shows the code for the `SIOCADDMULTI` and `SIOCDELMULTI` options.

```

-----if_le.c
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

```

Figure 12.31 `leioctl` function: multicast processing.

657-671 `leioctl` passes add and delete requests directly to the `ether_addmulti` or `ether_delmulti` functions. Both functions return `ENETRESET` if the request changes the set of IP multicast addresses that must be received by the physical hardware. If this occurs, `leioctl` calls `lereset` to reinitialize the hardware with the new multicast reception list.

We don't show `lereset`, as it is specific to the LANCE Ethernet hardware. For multicasting, `lereset` arranges for the hardware to receive frames addressed to any of the Ethernet multicast addresses contained in the `ether_multi` list associated with the interface. The LANCE driver uses a hashing mechanism if each entry on the multicast list is a single address. The hash code allows the hardware to receive multicast packets selectively. If the driver finds an entry that describes a range of addresses, it abandons the hash strategy and configures the hardware to receive *all* multicast packets. If the driver must fall back to receiving all Ethernet multicast addresses, the `IFF_ALLMULTI` flag is on when `lereset` returns.

### `ether_addmulti` Function

Every Ethernet driver calls `ether_addmulti` to process the `SIOCADDMULTI` request. This function maps the IP class D address to the appropriate Ethernet multicast address (Figure 12.5) and updates the `ether_multi` list. Figure 12.32 shows the first half of the `ether_addmulti` function.

#### Initialize address range

366-399 First, `ether_addmulti` initializes a range of multicast addresses in `addrlo` and `addrhi` (both are arrays of six unsigned characters). If the requested address is from the `AF_UNSPEC` family, `ether_addmulti` assumes the address is an explicit Ethernet multicast address and copies it into `addrlo` and `addrhi`. If the address is in the `AF_INET` family and is `INADDR_ANY` (0.0.0.0), `ether_addmulti` initializes `addrlo` to `ether_ipmulticast_min` and `addrhi` to `ether_ipmulticast_max`. These two constant Ethernet addresses are defined as:

```
u_char ether_ipmulticast_min[6] = { 0x01, 0x00, 0x5e, 0x00, 0x00, 0x00 };
u_char ether_ipmulticast_max[6] = { 0x01, 0x00, 0x5e, 0x7f, 0xff, 0xff };
```

As with `etherbroadcastaddr` (Section 4.3), this is a convenient way to define a 48-bit constant.

IP multicast routers must listen for all IP multicasts. Specifying the group as `INADDR_ANY` is considered a request to join *every* IP multicast group. The Ethernet address range selected in this case spans the entire block of IP multicast addresses allocated to the IANA.

The `mroute(8)` daemon issues a `SIOCADDMULTI` request with `INADDR_ANY` when it begins routing packets for a multicast interface.

`ETHER_MAP_IP_MULTICAST` maps any other specific IP multicast group to the appropriate Ethernet multicast address. Requests for other address families are rejected with an `EAFNOSUPPORT` error.

While the Ethernet multicast list supports address ranges, there is no way for a process or the kernel to request a specific range, other than to enumerate the addresses, since `addrlo` and `addrhi` are always set to the same address.

The second half of `ether_addmulti`, shown in Figure 12.33, verifies the address range and adds it to the list if it is new.

```

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     switch (ifr->ifr_addr.sa_family) {

377     case AF_UNSPEC:
378         bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
379         bcopy(addrlo, addrhi, 6);
380         break;

381     case AF_INET:
382         sin = (struct sockaddr_in *) &(ifr->ifr_addr);
383         if (sin->sin_addr.s_addr == INADDR_ANY) {
384             /*
385              * An IP address of INADDR_ANY means listen to all
386              * of the Ethernet multicast addresses used for IP.
387              * (This is for the sake of IP multicast routers.)
388              */
389             bcopy(ether_ipmulticast_min, addrlo, 6);
390             bcopy(ether_ipmulticast_max, addrhi, 6);
391         } else {
392             ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
393             bcopy(addrlo, addrhi, 6);
394         }
395         break;

396     default:
397         splx(s);
398         return (EAFNOSUPPORT);
399     }
}

```

*if\_ethersubr.c*

*if\_ethersubr.c*

Figure 12.32 ether\_addmulti function: first half.

**Already receiving**

400-418 ether\_addmulti checks the multicast bit (Figure 4.12) of the high and low addresses to ensure that they are indeed Ethernet multicast addresses. ETHER\_LOOKUP\_MULTI (Figure 12.9) determines if the hardware is already listening for the specified multicast addresses. If so, the reference count (enm\_refcount) in the matching ether\_multi structure is incremented and ether\_addmulti returns 0.

**Update ether\_multi list**

419-441 If this is a new address range, a new ether\_multi structure is allocated, initialized, and linked to the ac\_multiaddrs list in the interface's arpcom structure (Figure 12.8). If ENETRESET is returned by ether\_addmulti, the device driver that called

```

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 }

```

Figure 12.33 ether\_addmulti function: second half.

the function knows that the multicast list has changed and the hardware reception filter must be updated.

Figure 12.34 shows the relationships between the `ip_moptions`, `in_multi`, and `ether_multi` structures after the LANCE Ethernet interface has joined the all-hosts group.

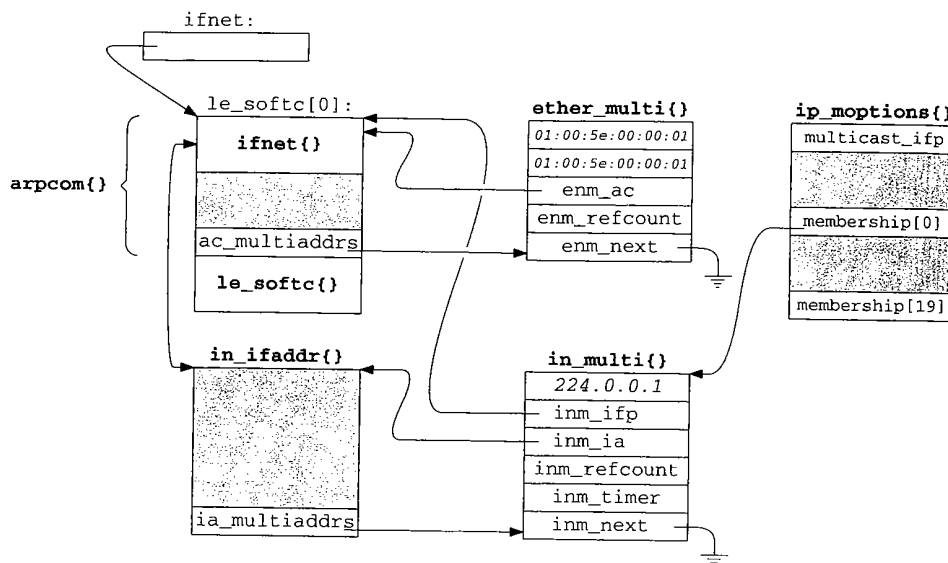


Figure 12.34 Overview of multicast data structures.

## 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_options` 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.

### 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.



```

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;

```

Figure 12.35 ip\_setsockopt function: leaving a multicast group.

**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.

```

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     if (--inm->inm_refcount == 0) {
542         /*
543          * No remaining claims to this record; let IGMP know that
544          * we are leaving the multicast group.
545          */
546         igmp_leavegroup(inm);
547         /*
548          * Unlink from list.
549          */
550         for (p = &inm->inm_ia->ia_multiaddrs;
551              *p != inm;
552              p = &(*p)->inm_next)
553             continue;
554         *p = (*p)->inm_next;
555         /*
556          * Notify the network driver to update its multicast reception
557          * filter.
558          */
559         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_family = AF_INET;
560         ((struct sockaddr_in *) &(ifr.ifr_addr))->sin_addr =
561             inm->inm_addr;
562         (*inm->inm_ifp->if_ioctl) (inm->inm_ifp, SIOCDELMULTI,
563                                 (caddr_t) & ifr);
564         free(inm, M_IPMADDR);
565     }
566     splx(s);
567 }

```

Figure 12.36 `in_delmulti` function.

**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.

445-479 `ether_delmulti` initializes the `addrlo` and `addrhi` arrays in the same way as `ether_admulti` 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.

```
if_ethersubr.c
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 = splmp();

456     switch (ifr->ifr_addr.sa_family) {

457     case AF_UNSPEC:
458         bcopy(ifr->ifr_addr.sa_data, addrlo, 6);
459         bcopy(addrlo, addrhi, 6);
460         break;

461     case AF_INET:
462         sin = (struct sockaddr_in *) &(ifr->ifr_addr);
463         if (sin->sin_addr.s_addr == INADDR_ANY) {
464             /*
465              * An IP address of INADDR_ANY means stop listening
466              * to the range of Ethernet multicast addresses used
467              * for IP.
468              */
469             bcopy(ether_ipmulticast_min, addrlo, 6);
470             bcopy(ether_ipmulticast_max, addrhi, 6);
471         } else {
472             ETHER_MAP_IP_MULTICAST(&sin->sin_addr, addrlo);
473             bcopy(addrlo, addrhi, 6);
474         }
475         break;

476     default:
477         splx(s);
478         return (EAFNOSUPPORT);
479     }

480     /*
481     * Look up the address in our list.
482     */
483     ETHER_LOOKUP_MULTI(addrlo, addrhi, ac, enm);
484     if (enm == NULL) {
485         splx(s);
486         return (ENXIO);
487     }
488     if (--enm->enm_refcount != 0) {
489         /*
490         * Still some claims to this record.
491         */
492         splx(s);
493         return (0);
494     }
}
```

```

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\_ethersubr.c*

Figure 12.37 ether\_delmulti function.

## 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.

### 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`, `t11`, 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.

```
876 int
877 ip_getmoptions(optname, imo, mp)
878 int optname;
879 struct ip_moptions *imo;
880 struct mbuf **mp;
881 {
882     u_char *ttl;
883     u_char *loop;
884     struct in_addr *addr;
885     struct in_ifaddr *ia;
886
887     *mp = m_get(M_WAIT, MT_SOOPTS);
888
889     switch (optname) {
890
891     case IP_MULTICAST_IF:
892         addr = mtod(*mp, struct in_addr *);
893         (*mp)->m_len = sizeof(struct in_addr);
894         if (imo == NULL || imo->imo_multicast_ifp == NULL)
895             addr->s_addr = INADDR_ANY;
896         else {
897             IFP_TO_IA(imo->imo_multicast_ifp, ia);
898             addr->s_addr = (ia == NULL) ? INADDR_ANY
899                 : IA_SIN(ia)->sin_addr.s_addr;
900         }
901         return (0);
902
903     case IP_MULTICAST_TTL:
904         ttl = mtod(*mp, u_char *);
905         (*mp)->m_len = 1;
906         *ttl = (imo == NULL) ? IP_DEFAULT_MULTICAST_TTL
907             : imo->imo_multicast_ttl;
908         return (0);
909
910     case IP_MULTICAST_LOOP:
911         loop = mtod(*mp, u_char *);
912         (*mp)->m_len = 1;
913         *loop = (imo == NULL) ? IP_DEFAULT_MULTICAST_LOOP
914             : imo->imo_multicast_loop;
915         return (0);
916
917     default:
918         return (EOPNOTSUPP);
919     }
920 }
```

Figure 12.38 ip\_getmoptions function.

`_output.c`

## 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.

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).

`_output.c`

```

214     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
215         struct in_multi *inm;
216         extern struct socket *ip_router;

217         if (ip_router) {
218             /*
219              * If we are acting as a multicast router, all
220              * incoming multicast packets are passed to the
221              * kernel-level multicast forwarding function.
222              * The packet is returned (relatively) intact; if
223              * ip_mforward() returns a non-zero value, the packet
224              * must be discarded, else it may be accepted below.
225              *
226              * (The IP ident field is put in the same byte order
227              * as expected when ip_mforward() is called from
228              * ip_output().)
229              */
230             ip->ip_id = htons(ip->ip_id);
231             if (ip_mforward(m, m->m_pkthdr.rcvif) != 0) {
232                 ipstat.ips_cantforward++;
233                 m_freem(m);
234                 goto next;
235             }
236             ip->ip_id = ntohs(ip->ip_id);

237             /*
238              * The process-level routing demon needs to receive
239              * all multicast IGMP packets, whether or not this
240              * host belongs to their destination groups.
241              */
242             if (ip->ip_p == IPPROTO_IGMP)
243                 goto ours;
244             ipstat.ips_forward++;
245         }
246         /*
247          * See if we belong to the destination multicast group on the
248          * arrival interface.
249          */
250         IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.rcvif, inm);
251         if (inm == NULL) {
252             ipstat.ips_cantforward++;
253             m_freem(m);
254             goto next;
255         }
256         goto ours;
257     }

```

Figure 12.39 ipintr function: multicast input processing.



nput.c

## 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.

```

129     if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
130         struct in_multi *inm;
131         extern struct ifnet loif;

132         m->m_flags |= M_MCAST;
133         /*
134          * IP destination address is multicast. Make sure "dst"
135          * still points to the address in "ro". (It may have been
136          * changed to point to a gateway address, above.)
137          */
138         dst = (struct sockaddr_in *) &ro->ro_dst;
139         /*
140          * See if the caller provided any multicast options
141          */
142         if (imo != NULL) {
143             ip->ip_ttl = imo->imo_multicast_ttl;
144             if (imo->imo_multicast_ifp != NULL)
145                 ifp = imo->imo_multicast_ifp;
146         } else
147             ip->ip_ttl = IP_DEFAULT_MULTICAST_TTL;
148         /*
149          * Confirm that the outgoing interface supports multicast.
150          */
151         if ((ifp->if_flags & IFF_MULTICAST) == 0) {
152             ipstat.ips_noroute++;
153             error = ENETUNREACH;
154             goto bad;
155         }
156         /*
157          * If source address not specified yet, use address
158          * of outgoing interface.
159          */
160         if (ip->ip_src.s_addr == INADDR_ANY) {
161             struct in_ifaddr *ia;

162             for (ia = in_ifaddr; ia; ia = ia->ia_next)
163                 if (ia->ia_ifp == ifp) {
164                     ip->ip_src = IA_SIN(ia)->sin_addr;
165                     break;
166                 }
167         }

```

Figure 12.40 `ip_output` function: defaults and source address.

\_input.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`.

```

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_mrouter;
191         if (ip_mrouter && (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 }

```

Figure 12.41 ip\_output function: loopback, forward, and send.

#### 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.

```

935 static void                                     ip_output.c
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 }

```

Figure 12.42 `ip_mloopback` function.

**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.

## 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.

## 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

- 12.1 What is the difference between sending an IP broadcast packet to 255.255.255.255 and sending an IP multicast to the all-hosts group 224.0.0.1?
- 12.2 Why are interfaces identified by their IP unicast addresses in the multicasting code? What must be changed so that an interface could send and receive multicast datagrams but not have a unicast IP address?
- 12.3 In Section 12.3 we said that 32 IP groups are mapped to a single Ethernet address. Since 9 bits of a 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?

- 12.4 Why do you think `IP_MAX_MEMBERSHIPS` is set to 20? Could it be set to a larger value? Hint: Consider the size of the `ip_options` structure (Figure 12.15).
- 12.5 What happens when a multicast datagram is looped back by IP and is also received by the hardware interface on which it is transmitted (i.e., a nonsimplex interface)?
- 12.6 Draw a picture of a network with a multihomed host so that a multicast packet sent on one interface may be received on the other interface even if the host is not acting as a multicast router.
- 12.7 Trace the membership add request through the SLIP and loopback interfaces instead of the Ethernet interface.
- 12.8 How could a process request that the kernel join more than `IP_MAX_MEMBERSHIPS`?
- 12.9 Computing the checksum on a looped back packet is superfluous. 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?
- 12.13 Why is the group membership mechanism separate from the binding mechanism used to receive unicast and broadcast datagrams?

ter 12

value?

by the

on one  
multicast

of the

nod to

multi-

ce has  
OK?

tion is

used to

# 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.

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_fasttimo`, which calls `igmp_sendreport` as each timer expires.

We saw in Chapter 12 that `ip_setmoptions` calls `igmp_joyingroup` when a new `in_multi` structure is created. `igmp_joyingroup` 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.

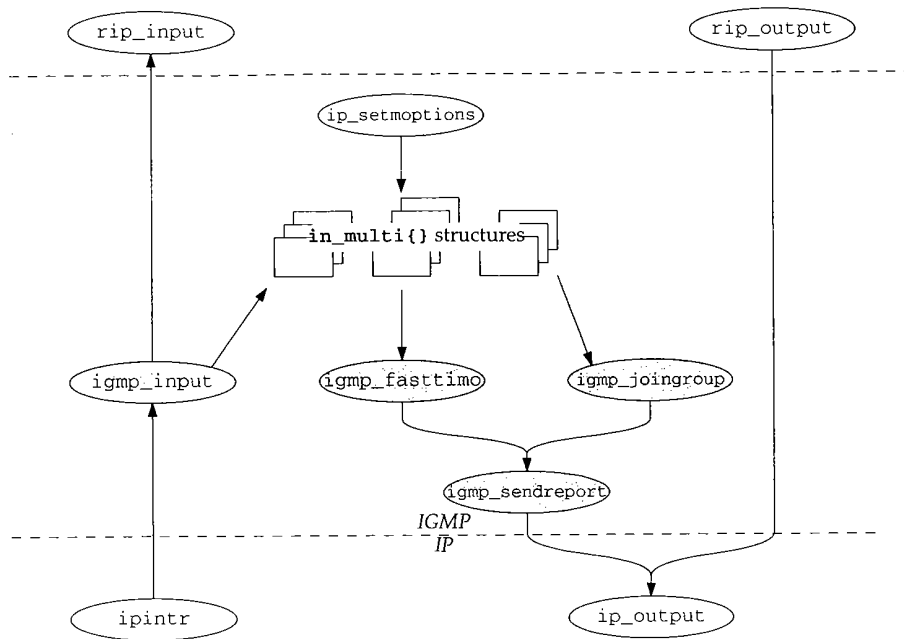


Figure 13.1 Summary of IGMP processing.

### 13.2 Code Introduction

The IGMP protocol is implemented in four files listed in Figure 13.2.

File	Description
<code>netinet/igmp.h</code>	IGMP protocol definitions
<code>netinet/igmp_var.h</code>	IGMP implementation definitions
<code>netinet/in_var.h</code>	IP multicast data structures
<code>netinet/igmp.c</code>	IGMP protocol implementation

Figure 13.2 Files discussed in this chapter.

#### Global Variables

Three new global variables, shown in Figure 13.3, are introduced in this chapter.

#### Statistics

IGMP statistics are maintained in the `igmpstat` variables shown in Figure 13.4.

SNMP



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).

Figure 13.3 Global variables introduced in this chapter.

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 13.4 IGMP statistics.

Figure 13.5 shows some sample output of these statistics, from the netstat -p igmp command on vangogh.cs.berkeley.edu.

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

Figure 13.5 Sample IGMP statistics.

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.

### 13.3 igmp Structure

An IGMP message is only 8 bytes long. Figure 13.6 shows the `igmp` structure used by Net/3.

```

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*

Figure 13.6 `igmp` structure.

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.

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)

Figure 13.7 IGMP message types.

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.

### 13.4 IGMP protosw Structure

Figure 13.9 describes the `protosw` structure for 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.

ed by

igmp.h

igmp.h

Fig-

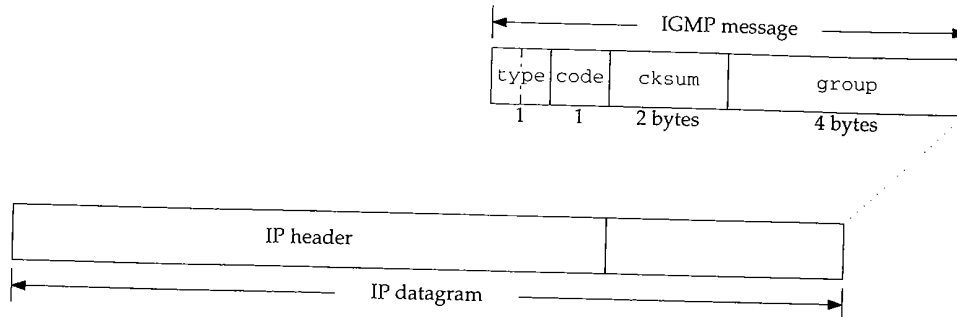


Figure 13.8 An IGMP message (igmp\_omitted).

pe 1  
hosts  
pe 2  
lticast  
ation  
under

Member	inetsw[5]	Description
pr_type	<i>SOCK_RAW</i>	IGMP provides raw packet services
pr_domain	<i>&amp;inetdomain</i>	IGMP is part of the Internet domain
pr_protocol	<i>IPPROTO_IGMP (2)</i>	appears in the <i>ip_p</i> field of the IP header
pr_flags	<i>PR_ATOMIC   PR_ADDR</i>	socket layer flags, not used by protocol processing
pr_input	<i>igmp_input</i>	receives messages from IP layer
pr_output	<i>rip_output</i>	sends IGMP message to IP layer
pr_ctlinput	<i>0</i>	not used by IGMP
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>igmp_init</i>	initialization for IGMP
pr_fasttimo	<i>igmp_fasttimo</i>	process pending membership reports
pr_slowtimo	<i>0</i>	not used by IGMP
pr_drain	<i>0</i>	not used by IGMP
pr_sysctl	<i>0</i>	not used by IGMP

Figure 13.9 The IGMP protosw structure.

iar IP  
being

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.

IGMP  
cesses  
a raw

### 13.5 Joining a Group: `igmp_joiningroup` Function

We saw in Chapter 12 that `igmp_joiningroup` 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_joiningroup` is not called. `igmp_joiningroup` is shown in Figure 13.10

```

164 void
165 igmp_joiningroup(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*

Figure 13.10 `igmp_joiningroup` function.

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_joiningroup` 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_fasttimo` (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).

59-73 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

$$1 \leq (r \bmod 50) + 1 \leq 50$$

```

59 /*
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.ips_total + \
69       ntohl(IA_SIN(in_ifaddr)->sin_addr.s_addr) + \
70       ntohl((multiaddr).s_addr) \
71       ) \
72       % (IGMP_MAX_HOST_REPORT_DELAY * PR_FASTHZ) + 1 \
73       )

```

Figure 13.11 IGMP\_RANDOM\_DELAY function.

Zero is avoided because it would disable the timer and no report would be sent.

### 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.

```

123 struct in_multistep {
124     struct in_ifaddr *i_ia;
125     struct in_multi *i_inm;
126 };

```

Figure 13.12 `in_multistep` function.

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.

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.

```

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 }

```

Figure 13.13 IN\_FIRST\_MULTI and IN\_NEXT\_MULTI structures.

We know from Figure 13.9 that `igmp_fasttimo` is the fast timeout function for IGMP and is called five times per second. `igmp_fasttimo` (shown in Figure 13.14) decrements multicast report timers and sends a report when the timer expires.

187-198 If `igmp_timers_are_running` is false, `igmp_fasttimo` returns immediately instead of wasting time examining each timer.

199-213 `igmp_fasttimo` resets the running flag and then initializes `step` and `inm` with `IN_FIRST_MULTI`. The `igmp_fasttimo` 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.

\_var.h

p"  
e

```

187 void
188 igmp_fasttimo()
189 {
190     struct in_multi *inm;
191     int s;
192     struct in_multistep step;
193
194     /*
195      * Quick check to see if any work needs to be done, in order
196      * to minimize the overhead of fasttimo processing.
197      */
198     if (!igmp_timers_are_running)
199         return;
200
201     s = splnet();
202     igmp_timers_are_running = 0;
203     IN_FIRST_MULTI(step, inm);
204     while (inm != NULL) {
205         if (inm->inm_timer == 0) {
206             /* do nothing */
207         } else if (--inm->inm_timer == 0) {
208             igmp_sendreport(inm);
209         } else {
210             igmp_timers_are_running = 1;
211         }
212         IN_NEXT_MULTI(step, inm);
213     }
214     splx(s);
215 }

```

igmp.c

Figure 13.14 igmp\_fasttimo function.

i\_var.h

**igmp\_sendreport Function**

The `igmp_sendreport` function (shown in Figure 13.15) constructs and sends an IGMP report message for a single multicast group.

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.

on for  
13.14)

liately

with  
cture

ership

ig, so

```
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     {
260         extern struct socket *ip_mrouter;
261         imo->imo_multicast_loop = (ip_mrouter != NULL);
262     }
263     ip_output(m, NULL, NULL, 0, imo);
264
265     ++igmpstat.igmps_snd_reports;
266 }
```

Figure 13.15 igmp\_sendreport function.



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).

### 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.

#### 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 `mrouterd` 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.

igmp.c

```

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
66     ++igmpstat.igps_rcv_total;
67
68     ip = mtod(m, struct ip *);
69     igmplen = ip->ip_len;
70
71     /*
72      * Validate lengths
73      */
74     if (igmplen < IGMP_MINLEN) {
75         ++igmpstat.igps_rcv_tooshort;
76         m_freem(m);
77         return;
78     }
79     minlen = iphlen + IGMP_MINLEN;
80     if ((m->m_flags & M_EXT || m->m_len < minlen) &&
81         (m = m_pullup(m, minlen)) == 0) {
82         ++igmpstat.igps_rcv_tooshort;
83         return;
84     }
85     /*
86      * Validate checksum
87      */
88     m->m_data += iphlen;
89     m->m_len -= iphlen;
90     igmp = mtod(m, struct igmp *);
91     if (in_cksum(m, igmplen)) {
92         ++igmpstat.igps_rcv_badsum;
93         m_freem(m);
94         return;
95     }
96     m->m_data -= iphlen;
97     m->m_len += iphlen;
98     ip = mtod(m, struct ip *);
99
100     switch (igmp->igmp_type) {
101
102         /* switch cases */
103
104     }
105 }

```

97

```

158  /*
159  * Pass all valid IGMP packets up to any process(es) listening
160  * on a raw IGMP socket.
161  */
162  rip_input(m);
163 )

```

igmp.c

Figure 13.16 igmp\_input function.

```

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

Figure 13.17 Input processing of the IGMP query message.

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.

```

123     case IGMP_HOST_MEMBERSHIP_REPORT:
124         ++igmpstat.igps_rcv_reports;

125         if (ifp == &loif)
126             break;

127         if (!IN_MULTICAST(ntohl(igmp->igmp_group.s_addr)) ||
128             igmp->igmp_group.s_addr != ip->ip_dst.s_addr) {
129             ++igmpstat.igps_rcv_badreports;
130             m_freem(m);
131             return;
132         }
133         /*
134          * KLUDGE: if the IP source address of the report has an
135          * unspecified (i.e., zero) subnet number, as is allowed for
136          * a booting host, replace it with the correct subnet number
137          * so that a process-level multicast routing demon can
138          * determine which subnet it arrived from. This is necessary
139          * to compensate for the lack of any way for a process to
140          * determine the arrival interface of an incoming packet.
141          */
142         if ((ntohl(ip->ip_src.s_addr) & IN_CLASSA_NET) == 0) {
143             IFP_TO_IA(ifp, ia);
144             if (ia)
145                 ip->ip_src.s_addr = htonl(ia->ia_subnet);
146         }
147         /*
148          * If we belong to the group being reported, stop
149          * our timer for that group.
150          */
151         IN_LOOKUP_MULTI(igmp->igmp_group, ifp, inm);
152         if (inm != NULL) {
153             inm->inm_timer = 0;
154             ++igmpstat.igps_rcv_ourreports;
155         }
156         break;

```

Figure 13.18 Input processing of the IGMP report message.

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.

### 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.

```
179 void  
180 igmp_leavegroup(inm) igmp.c  
181 struct in_multi *inm;  
182 {  
183     /*  
184     * No action required on leaving a group.  
185     */  
186 }
```

Figure 13.19 `igmp_leavegroup` function. igmp.c

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.

## 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

- 13.1 Why isn't it necessary to respond to an IGMP query on the loopback interface?
- 13.2 Verify the assumption stated on lines 226 to 229 in Figure 13.15.
- 13.3 Is it necessary to set random delays for membership queries that arrive on a point-to-point network interface?

# 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 `mrouterd` 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 `mrouterd` and how they correspond to the BSD releases. The `mrouterd` releases include both the user-level daemons and the kernel-level multicast code.

mrouterd version	Description
1.2	modifies the 4.3BSD Tahoe release
2.0	included with 4.4BSD and Net/3
3.3	modifies SunOS 4.1.3

Figure 14.1 `mrouterd` and IP multicasting releases.

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.

## 14.2 Code Introduction

The three files listed in Figure 14.2 are 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

Figure 14.2 Files discussed in this chapter.

### Global Variables

The global variables used by the multicast routing code are shown in Figure 14.3.

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

Figure 14.3 Global variables introduced in this chapter.

### 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.



works  
s over  
e con-  
roved  
data-  
ed by  
  
ocols  
l that

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 14.4 Statistics collected in this chapter.

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

Figure 14.5 Sample IP multicast routing statistics.

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.

## 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.

t struc-  
atistics,

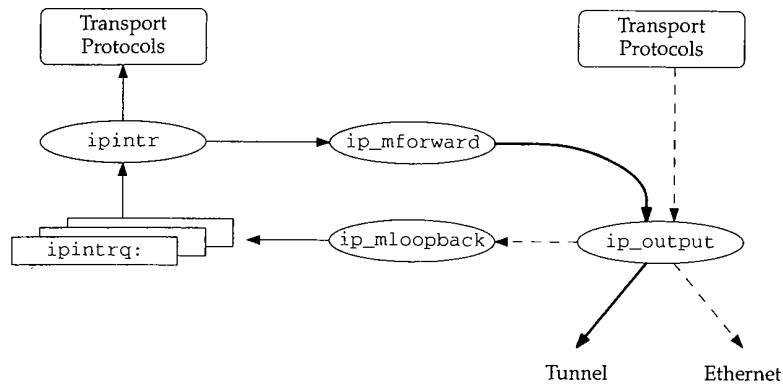


Figure 14.6 Multicast output processing with loopback.

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.

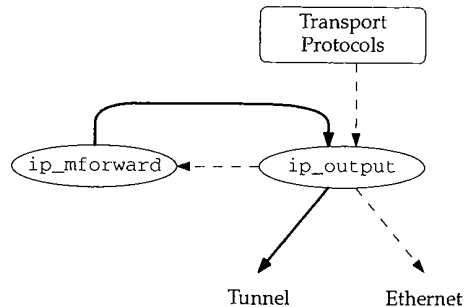


Figure 14.7 Multicast output processing with no loopback.

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.

## 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 `mrouted`—we 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.

optname	optval type	Function	Description
<code>DVMRP_INIT</code>		<code>ip_mrouter_init</code>	<code>mrouted</code> is starting
<code>DVMRP_DONE</code>		<code>ip_mrouter_done</code>	<code>mrouted</code> is shutting down
<code>DVMRP_ADD_VIF</code>	<code>struct vifctl</code>	<code>add_vif</code>	add virtual interface
<code>DVMRP_DEL_VIF</code>	<code>vifi_t</code>	<code>del_vif</code>	delete virtual interface
<code>DVMRP_ADD_LGRP</code>	<code>struct lgrpctl</code>	<code>add_lgrp</code>	add multicast group entry for an interface
<code>DVMRP_DEL_LGRP</code>	<code>struct lgrpctl</code>	<code>del_lgrp</code>	delete multicast group entry for an interface
<code>DVMRP_ADD_MRT</code>	<code>struct mrtctl</code>	<code>add_mrt</code>	add multicast route
<code>DVMRP_DEL_MRT</code>	<code>struct in_addr</code>	<code>del_mrt</code>	delete multicast route

Figure 14.8 Multicast routing socket options.

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.

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.

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.

```

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);

```

Figure 14.9 rip\_ctloutput function: DVMRP\_xxx socket options.

```

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
92     if (cmd != DVMRP_INIT && so != ip_mrouter)
93         error = EACCES;
94     else
95         switch (cmd) {
96             case DVMRP_INIT:
97                 error = ip_mrouter_init(so);
98                 break;
99             case DVMRP_DONE:
100                 error = ip_mrouter_done();
101                 break;
102             case DVMRP_ADD_VIF:
103                 if (m == NULL || m->m_len < sizeof(struct vifctl))
104                     error = EINVAL;
105                 else
106                     error = add_vif(mtod(m, struct vifctl *));
107                 break;
108             case DVMRP_DEL_VIF:
109                 if (m == NULL || m->m_len < sizeof(short))
110                     error = EINVAL;
111                 else
112                     error = del_vif(mtod(m, vifi_t *));
113                 break;

```

```

113     case DVMRP_ADD_LGRP:
114         if (m == NULL || m->m_len < sizeof(struct lgrplctl))
115             error = EINVAL;
116         else
117             error = add_lgrp(mtod(m, struct lgrplctl *));
118         break;

119     case DVMRP_DEL_LGRP:
120         if (m == NULL || m->m_len < sizeof(struct lgrplctl))
121             error = EINVAL;
122         else
123             error = del_lgrp(mtod(m, struct lgrplctl *));
124         break;

125     case DVMRP_ADD_MRT:
126         if (m == NULL || m->m_len < sizeof(struct mrtctl))
127             error = EINVAL;
128         else
129             error = add_mrt(mtod(m, struct mrtctl *));
130         break;

131     case DVMRP_DEL_MRT:
132         if (m == NULL || m->m_len < sizeof(struct in_addr))
133             error = EINVAL;
134         else
135             error = del_mrt(mtod(m, struct in_addr *));
136         break;

137     default:
138         error = EOPNOTSUPP;
139         break;
140     }
141     return (error);
142 }

```

*ip\_mroute.c*Figure 14.10 *ip\_mrouter\_cmd* function.

84-92 The first command issued by *mrouterd* 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 *mrouterd* issues the *DVMRP\_INIT* command during initialization.

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 *mrouterd*.

```

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
154     if (ip_mrouter != NULL)
155         return (EADDRINUSE);
156
157     ip_mrouter = so;
158
159     return (0);
160 }

```

ip\_mroute.c

Figure 14.11 ip\_mrouter\_init function: DVMRP\_INIT command.

The remainder of the DVMRP\_xxx commands are described in the following sections.

## 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.

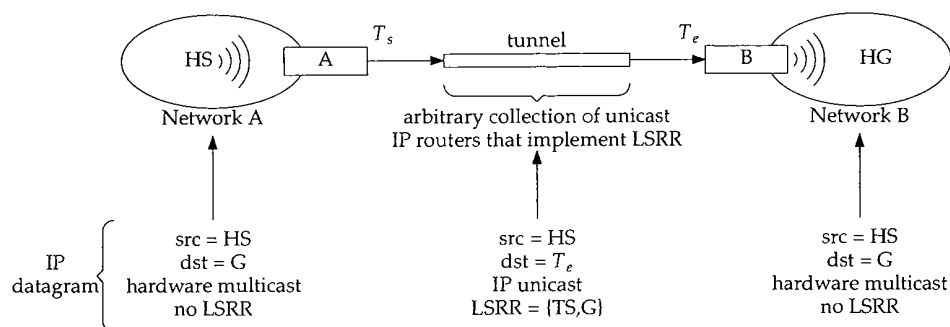


Figure 14.12 A multicast tunnel.

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.

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	$T_s$ see text $\bullet$	after ip_doptions on router B
$T_e$	HS	G			after ip_mforward on router B

Figure 14.13 LSRR multicast tunnel options.

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_doptions` on router B. Recall from Chapter 9 that `ip_doptions` 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_doptions` 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_doptions` 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 `mouted`, 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. LSR tunneling is supported in newer versions of `mroute` 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.

```

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`

----- `ip_mroute.h`

Figure 14.14 `vif` structure.

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.



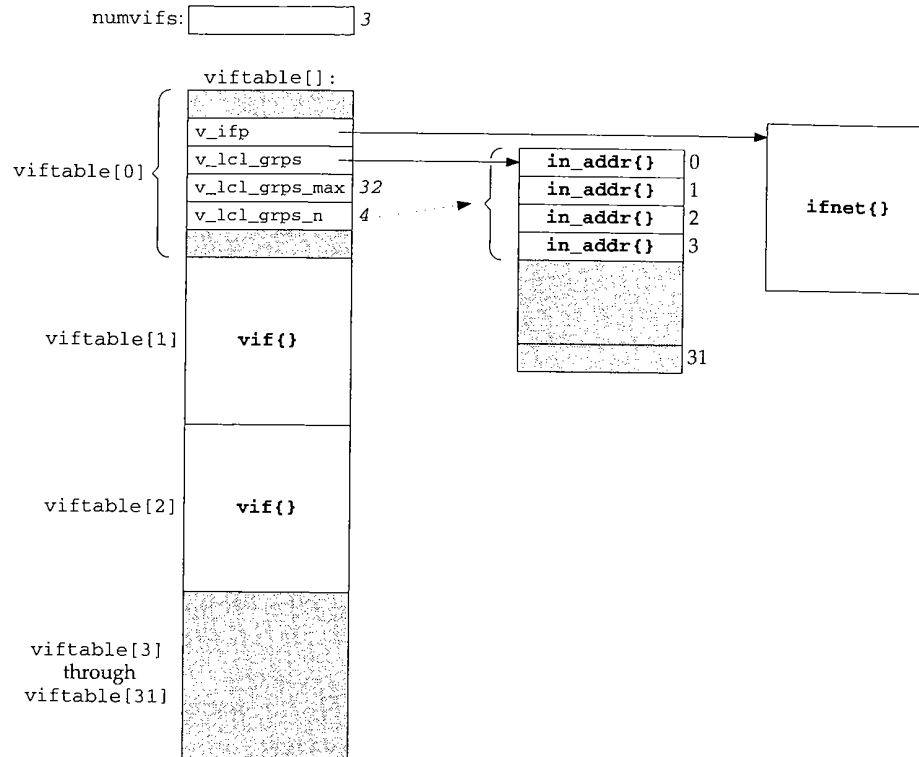


Figure 14.15 viftable array.

mroute 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 mroute begins. Multicast tunnels are added when mroute reads its configuration file, usually /etc/mroute.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 mroute to the kernel with the DVMRP\_ADD\_VIF command. It instructs the kernel to add an interface to the table of virtual interfaces.

```

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 };

```

ip\_mroute.h

ip\_mroute.h

Figure 14.16 vifctl structure.

76-82 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.

```

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 (vifcp->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         vifcp->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         satoisin(&ifr.ifr_addr)->sin_family = AF_INET;
241         satoisin(&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 }

```

*ip\_mroute.c*

```

244     vifp->v_flags = vifcp->vifc_flags;
245     vifp->v_threshold = vifcp->vifc_threshold;
246     vifp->v_lcl_addr = vifcp->vifc_lcl_addr;
247     vifp->v_ifp = ifa->ifa_ifp;

248     /* Adjust numvifs up if the vifi is higher than numvifs */
249     if (numvifs <= vifcp->vifc_vifi)
250         numvifs = vifcp->vifc_vifi + 1;

251     splx(s);
252     return (0);
253 }

```

*ip\_mroute.c*

Figure 14.17 `add_vif` function: `DVMRP_ADD_VIF` command.

#### 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.

#### 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.

```

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 }

```

Figure 14.18 del\_vif function: DVMRP\_DEL\_VIF command.

#### 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.

## 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 `lgrplctl` (Figure 14.19) structure is passed with both commands.

```

87 struct lgrplctl {
88     vifi_t  lgc_vifi;
89     struct in_addr lgc_gaddr;
90 };

```

ip\_mroute.h

ip\_mroute.h

Figure 14.19 `lgrplctl` structure.

87-90 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.

### 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.

#### 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.

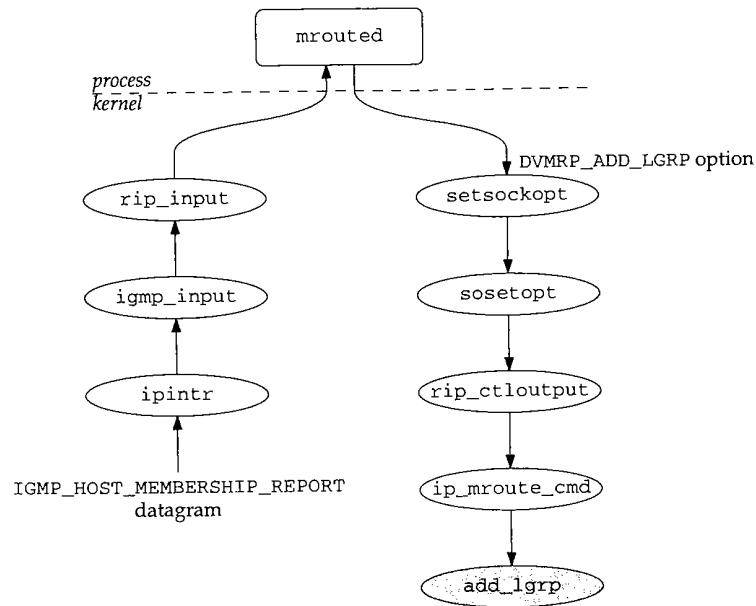


Figure 14.20 IGMP report processing.

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. `mrouded` maintains the appropriate timers and issues the `DVMRP_DEL_LGRP` command when the information expires. Figure 14.22 shows `del_lgrp`.

```

291 static int
292 add_lgrp(gcp)
293 struct lgrpctl *gcp;
294 {
295     struct vif *vifp;
296     int s;

297     if (gcp->lgc_vifi >= numvifs)
298         return (EINVAL);

299     vifp = viftable + gcp->lgc_vifi;
300     if (vifp->v_lcl_addr.s_addr == 0 || (vifp->v_flags & VIFF_TUNNEL))
301         return (EADDRNOTAVAIL);

302     /* If not enough space in existing list, allocate a larger one */
303     s = splnet();
304     if (vifp->v_lcl_grps_n + 1 >= vifp->v_lcl_grps_max) {
305         int num;
306         struct in_addr *ip;

307         num = vifp->v_lcl_grps_max;
308         if (num <= 0)
309             num = 32;          /* initial number */
310         else
311             num += num;       /* double last number */
312         ip = (struct in_addr *) malloc(num * sizeof(*ip),
313                                     M_MRTABLE, M_NOWAIT);
314         if (ip == NULL) {
315             splx(s);
316             return (ENOBUFS);
317         }
318         bzero((caddr_t) ip, num * sizeof(*ip)); /* XXX paranoid */
319         bcopy((caddr_t) vifp->v_lcl_grps, (caddr_t) ip,
320             vifp->v_lcl_grps_n * sizeof(*ip));

321         vifp->v_lcl_grps_max = num;
322         if (vifp->v_lcl_grps)
323             free(vifp->v_lcl_grps, M_MRTABLE);
324         vifp->v_lcl_grps = ip;

325         splx(s);
326     }
327     vifp->v_lcl_grps[vifp->v_lcl_grps_n++] = gcp->lgc_gaddr;

328     if (gcp->lgc_gaddr.s_addr == vifp->v_cached_group)
329         vifp->v_cached_result = 1;

330     splx(s);
331     return (0);
332 }

```

Figure 14.21 add\_lgrp function: process DVMRP\_ADD\_LGRP command.

```

337 static int
338 del_lgrp(gcp)
339 struct lgrplctl *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*

*ip\_mroute.c*

Figure 14.22 del\_lgrp function: process DVMRP\_DEL\_LGRP command.

#### 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.



ute.c

**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.

```

368 static int
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 }

```

*ip\_mroute.c*

*ip\_mroute.c*

route.c

ace is

**Figure 14.23** `grplst_member` function.**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.

mber-  
ice. If  
eared  
down

other-

## 14.7 Multicast Routing

As we mentioned at the start of this chapter, we will not be presenting the TRPB algorithm implemented by `mrouterd`, but we do need to provide a general overview of the mechanism to describe the multicast routing table and the multicast routing functions in the kernel. Figure 14.24 shows the sample multicast network that we use to illustrate the algorithms.

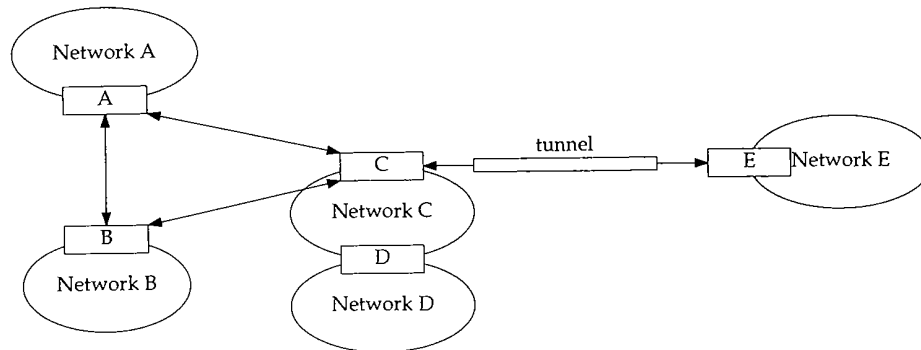


Figure 14.24 Sample multicast network.

In Figure 14.24, routers are shown as boxes and the ellipses are the multicast networks attached to the routers. For example, router D can multicast on network D and C. Router C can multicast to network C, to routers A and B through point-to-point interfaces, and to E through a multicast tunnel.

The simplest approach to multicast routing is to select a subset of the internet topology that forms a *spanning tree*. If each router forwards multicasts along the spanning tree, every router eventually receives the datagram. Figure 14.25 shows one spanning tree for our sample network, where host S on network A represents the source of a multicast datagram.

For a discussion of spanning trees, see [Tanenbaum 1989] or [Perlman 1992].

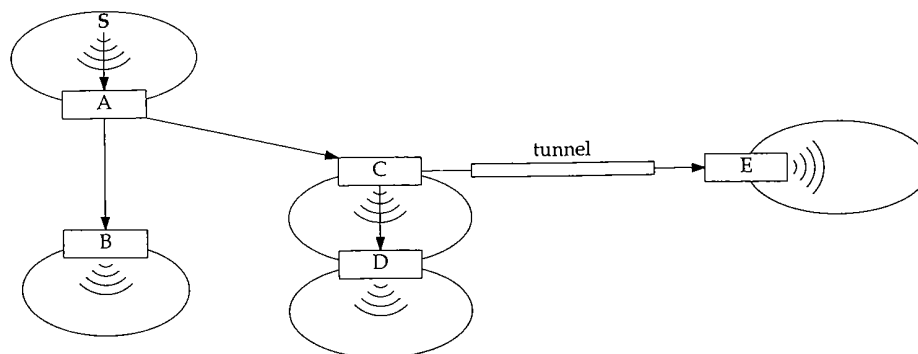


Figure 14.25 Spanning tree for network A.

We constructed the tree based on the shortest *reverse path* from every network back to the source in network A. In Figure 14.25, the link between routers B and C is omitted to form the spanning tree. The arrows between the source and router A, and between router C and D, emphasize that the multicast network is part of the spanning tree.

If the same spanning tree were used to forward a datagram from network C, the datagram would be forwarded along a longer path than needed to get to a recipient on network B. The algorithm described in RFC 1075 computes a separate spanning tree for each potential source network to avoid this problem. The routing tables contain a network number and subnet mask for each route, so that a single route applies to any host within the source subnet.

Because each spanning tree is constructed to provide the shortest reverse path to the source of the datagram, and every network receives every multicast datagram, this process is called *reverse path broadcasting* or RPB.

The RPB protocol has no knowledge of multicast group membership, so many datagrams are unnecessarily forwarded to networks that have no members in the destination group. If, in addition to computing the spanning trees, the routing algorithm records which networks are *leaves* and is aware of the group membership on each network, then routers attached to leaf networks can avoid forwarding datagrams onto the network when there is no member of the destination group present. This is called *truncated reverse path broadcasting* (TRPB), and is implemented by version 2.0 of mroute with the help of IGMP to keep track of membership in the leaf networks.

Figure 14.26 shows TRPB applied to a multicast sent from a source on network C and with a member of the destination group on network B.

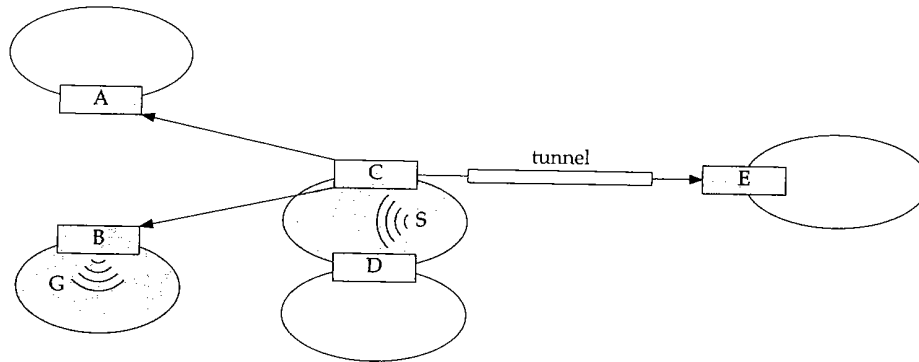


Figure 14.26 TRPB routing for network C.

We'll use Figure 14.26 to illustrate the terms used in the Net/3 multicast routing table. In this example, the shaded networks and routers receive a copy of the multicast datagram sent from the source on network C. The link between A and B is not part of the spanning tree and C does not have a link to D, since the multicast sent by the source is received directly by C and D.

In this figure, networks A, B, D, and E are leaf networks. Router C receives the multicast and forwards it through the interfaces attached to routers A, B, and E—even

though sending it to A and E is wasted effort. This is a major weakness of the TRPB algorithm.

The interface associated with network C on router C is called the *parent* because it is the interface on which router C expects to receive multicasts originating from network C. The interfaces from router C to routers A, B, and E, are *child* interfaces. For router A, the point-to-point interface is the parent for the source packets from C and the interface for network A is a child. Interfaces are identified as a parent or as a child relative to the source of the datagram. Multicast datagrams are forwarded only to the associated child interfaces, and never to the parent interface.

Continuing with the example, networks A, D, and E are not shaded because they are leaf networks without members of the destination group, so the spanning tree is truncated at the routers and the datagram is not forwarded onto these networks. Router B forwards the datagram onto network B, since there is a member of the destination group on the network. To implement the truncation algorithm, each multicast router that receives the datagram consults the group table associated with every virtual interface in the router's *viftable*.

The final refinement to the multicast routing algorithm is called *reverse path multicasting* (RPM). The goal of RPM is to *prune* each 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 a datagram to A and E, since there is no member of the destination group in those branches of the tree. Version 3.3 of *mrouterd* implements RPM.

Figure 14.27 shows our example network, but this time only the routers and networks reached when the datagram is routed by RPM are shaded.

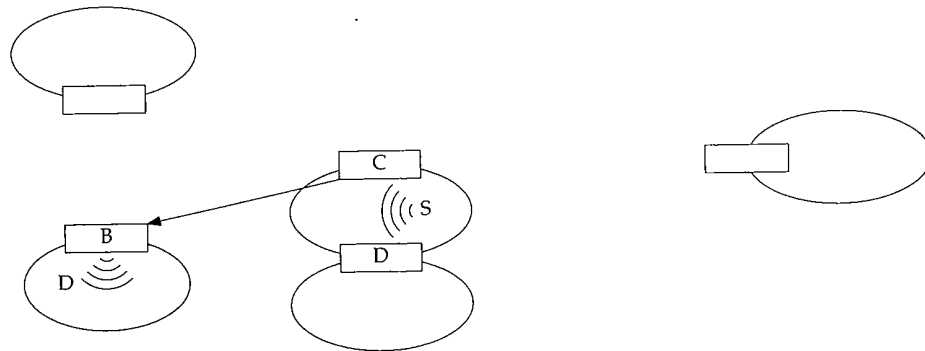


Figure 14.27 RPM routing for network C.

To compute the routing tables corresponding to the spanning trees we described, the multicast routers communicate with adjacent multicast routers to discover the multicast internet topology and the location of multicast group members. In Net/3, DVMRP is used for this communication. DVMRP messages are transmitted as IGMP datagrams and are sent to the multicast group 224.0.0.4, which is reserved for DVMRP communication (Figure 12.1).

In Figure 12.39, we saw that incoming IGMP packets are always accepted by a

multicast router. They are passed to `igmp_input`, to `rip_input`, and then read by `mROUTED` on a raw IGMP socket. `mROUTED` sends DVMRP messages to other multicast routers on the same raw IGMP socket.

For more information about RPB, TRPB, RPM, and the DVMRP messages that are needed to implement these algorithms, see [Deering and Cheriton 1990] and the source code release of `mROUTED`.

There are other multicast routing protocols in use on the Internet. Proteon routers implement the MOSPF protocol described in RFC 1584 [Moy 1994]. PIM (Protocol Independent Multicasting) is implemented by Cisco routers, starting with Release 10.2 of their operating software. PIM is described in [Deering et al. 1994].

### Multicast Routing Table

We can now describe the implementation of the multicast routing tables in Net/3. The kernel's multicast routing table is maintained as a hash table with 64 entries (`MRTHASHSIZ`). The table is kept in the global array `mrttable`, and each entry points to a linked list of `mrt` structures, shown in Figure 14.28.

```

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*

*ip\_mroute.h*

Figure 14.28 mrt structure.

`mrtc_origin` and `mrtc_originmask` identify an entry in the table. `mrtc_parent` is the index of the virtual interface on which all multicast datagrams from the origin are expected. The outgoing interfaces are identified within `mrtc_children`, which is a bitmap. Outgoing interfaces that are also leaves in the multicast routing tree are identified in `mrtc_leaves`, which is also a bitmap. The last member, `mrtc_next`, implements a linked list in case multiple routes hash to the same array entry.

Figure 14.29 shows the organization of the multicast routing table. Each `mrt` structure is placed in the hash chain that corresponds to return value from the `nethash` function shown in Figure 14.31.

The multicast routing table maintained by the kernel is a subset of the routing table maintained within `mROUTED` and contains enough information to support multicast forwarding within the kernel. Updates to the kernel table are sent with the `DVMRP_ADD_MRT` command, which includes the `mrtctl` structure shown in Figure 14.30.

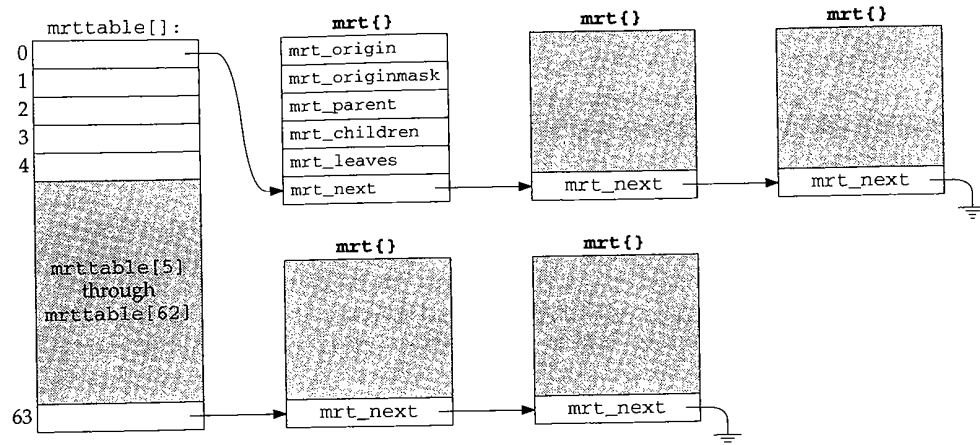


Figure 14.29 Multicast routing table.

```

95 struct mrtctl {
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

Figure 14.30 mrtctl structure.

95-101 The five members of the mrtctl structure carry the information we have already described (Figure 14.28) between mrtouted and the kernel.

The multicast routing table is keyed by the source IP address of the multicast datagram. nethash (Figure 14.31) implements the hashing algorithm used for the table. It accepts the source IP address and returns a value between 0 and 63 (MRTHASHSIZ - 1).

```

398 static u_long
399 nethash(in)
400 struct in_addr in;
401 {
402     u_long n;
403
404     n = in_netof(in);
405     while ((n & 0xff) == 0)
406         n >>= 8;
407     return (MRTHASHMOD(n));

```

ip\_mroute.c

Figure 14.31 nethash function.

398-407 `in_netof` returns `in` with the host portion set to all 0s leaving only the class A, B, or C network of the sending host in `n`. The result is shifted to the right until the low-order 8 bits are nonzero. `MRTHASHMOD` is

```
#define MRTHASHMOD(h) ((h) & (MRTHASHSIZ - 1))
```

The low-order 8 bits are logically ANDed with 63, leaving only the low-order 6 bits, which is an integer in the range 0 to 63.

Doing two function calls (`nethash` and `in_netof`) to calculate a hash value is an expensive algorithm to compute a hash for a 32-bit address.

### del\_mrt Function

The `mrtouted` daemon adds and deletes entries in the kernel's multicast routing table through the `DVMRP_ADD_MRT` and `DVMRP_DEL_MRT` commands. Figure 14.32 shows the `del_mrt` function.

```

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 = mrttable[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         mrttable[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*

*ip\_mroute.c*

Figure 14.32 `del_mrt` function: process `DVMRP_DEL_MRT` command.

### Find route entry

451-462 The for loop starts at the entry identified by `hash` (initialized in its declaration from `nethash`). If the entry is not located, `ESRCH` is returned.

**Delete route entry**

463-473 If the entry was stored in the cache, the cache is invalidated. The entry is unlinked from the hash chain and released. The `if` statement is needed to handle the special case when the matched entry is at the front of the list.

**add\_mrt Function**

The `add_mrt` function is shown in Figure 14.33.

```

411 static int
412 add_mrt(mrtcp)
413 struct mrtctl *mrtcp;
414 {
415     struct mrt *rt;
416     u_long hash;
417     int s;

418     if (rt = mrtfind(mrtcp->mrtc_origin)) {
419         /* Just update the route */
420         s = splnet();
421         rt->mrt_parent = mrtcp->mrtc_parent;
422         VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
423         VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
424         splx(s);
425         return (0);
426     }
427     s = splnet();

428     rt = (struct mrt *) malloc(sizeof(*rt), M_MRTABLE, M_NOWAIT);
429     if (rt == NULL) {
430         splx(s);
431         return (ENOBUFS);
432     }
433     /*
434      * insert new entry at head of hash chain
435      */
436     rt->mrt_origin = mrtcp->mrtc_origin;
437     rt->mrt_originmask = mrtcp->mrtc_originmask;
438     rt->mrt_parent = mrtcp->mrtc_parent;
439     VIFM_COPY(mrtcp->mrtc_children, rt->mrt_children);
440     VIFM_COPY(mrtcp->mrtc_leaves, rt->mrt_leaves);
441     /* link into table */
442     hash = nethash(mrtcp->mrtc_origin);
443     rt->mrt_next = mrttable[hash];
444     mrttable[hash] = rt;

445     splx(s);
446     return (0);
447 }

```

*ip\_mroute.c*

*ip\_mroute.c*

Figure 14.33 `add_mrt` function: process `DVMRP_ADD_MRT` command.



**Update existing route**

411-427 If the requested route is already in the routing table, the new information is copied into the route and `add_mrt` returns.

**Allocate new route**

428-447 An `mrt` structure is constructed in a newly allocated mbuf with the information from `mrtctl` structure passed with the add request. The hash index is computed from `mrtc_origin`, and the new route 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`, which returns a pointer to the matching `mrt` structure, or a null pointer if there is no match.

```

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
485     mrtstat.mrts_mrt_lookups++;
486
487     if (cached_mrt != NULL &&
488         (origin.s_addr & cached_originmask) == cached_origin)
489         return (cached_mrt);
490
491     mrtstat.mrts_mrt_misses++;
492
493     hash = nethash(origin);
494     for (rt = mrttable[hash]; rt; rt = rt->mrt_next)
495         if ((origin.s_addr & rt->mrt_originmask.s_addr) ==
496             rt->mrt_origin.s_addr) {
497             s = splnet();
498             cached_mrt = rt;
499             cached_origin = rt->mrt_origin.s_addr;
500             cached_originmask = rt->mrt_originmask.s_addr;
501             splx(s);
502             return (rt);
503         }
504     return (NULL);
505 }

```

Figure 14.34 `mrtfind` function.

**Check route lookup cache**

477-488 The given source IP address (`origin`) is logically ANDed with the origin mask in the cache. If the result matches `cached_origin`, the cached entry is returned.

**Check the hash table**

489-501 nethash returns the hash index for the route entry. The for loop searches the hash chain for a matching route. When a match is found, the cache is updated and a pointer to the route is returned. If a match is not found, a null pointer is returned.

**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.

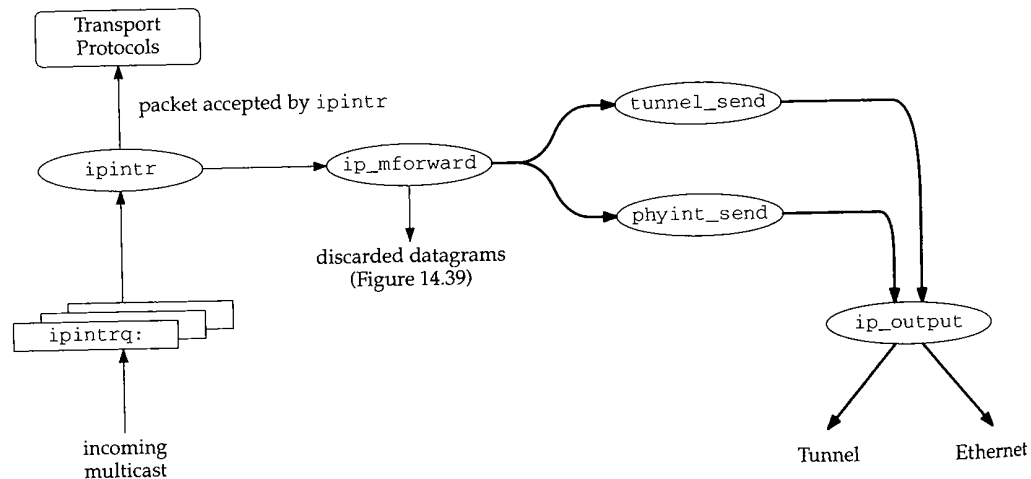


Figure 14.35 Multicast datagram arriving on physical interface.

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.

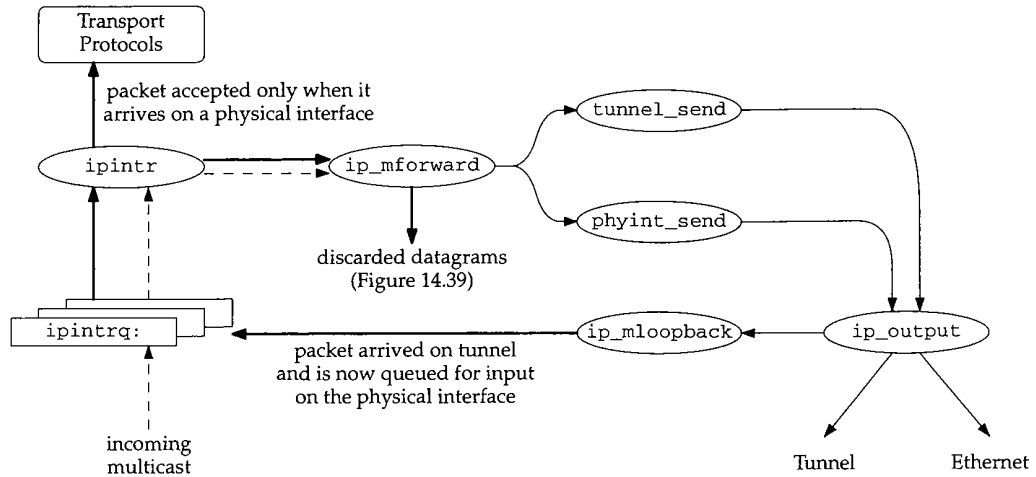


Figure 14.36 Multicast datagram arriving on a multicast tunnel.

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.

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:

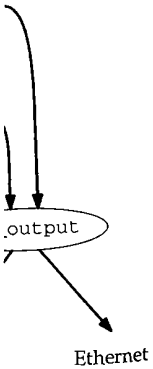
- tunnel input processing (Figure 14.37),
- forwarding eligibility (Figure 14.39), and
- forward to outgoing interfaces (Figure 14.40).

hes the d and a l.

2.39 that outer is

ams that n the one

an inter- er and is e system datagram et by the



r of the des- out process- licated and

```

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     if (ip->ip_hl < (IP_HDR_LEN + TUNNEL_LEN) >> 2 ||
528         (ipoptions = (u_char *) (ip + 1))[1] != IPOPT_LSRR) {
529         /* Packet arrived via a physical interface. */
530         tunnel_src = 0;
531     } else {
532         /*
533          * Packet arrived through a tunnel.
534          * A tunneled packet has a single NOP option and a
535          * two-element loose-source-and-record-route (LSRR)
536          * option immediately following the fixed-size part of
537          * the IP header. At this point in processing, the IP
538          * header should contain the following IP addresses:
539          *
540          * original source - in the source address field
541          * destination group - in the destination address field
542          * remote tunnel end-point - in the first element of LSRR
543          * one of this host's adrs - in the second element of LSRR
544          *
545          * NOTE: RFC-1075 would have the original source and
546          * remote tunnel end-point addresses swapped. However,
547          * that could cause delivery of ICMP error messages to
548          * innocent applications on intermediate routing
549          * hosts! Therefore, we hereby change the spec.
550          */
551         /* Verify that the tunnel options are well-formed. */
552         if (ipoptions[0] != IPOPT_NOP ||
553             ipoptions[2] != 11 || /* LSRR option length */
554             ipoptions[3] != 12 || /* LSRR address pointer */
555             (tunnel_src = *(u_long *) (&ipoptions[4])) == 0) {
556             mrtstat.mrts_bad_tunnel++;
557             return (1);
558         }
559         /* Delete the tunnel options from the packet. */
560         ovbcopy((caddr_t) (ipoptions + TUNNEL_LEN), (caddr_t) ipoptions,
561             (unsigned) (m->m_len - (IP_HDR_LEN + TUNNEL_LEN)));
562         m->m_len -= TUNNEL_LEN;
563         ip->ip_len -= TUNNEL_LEN;
564         ip->ip_hl -= TUNNEL_LEN >> 2;
565     }

```

ip\_mroute.c

Figure 14.37 ip\_mforward function: tunnel arrival.

ite.c

516-526 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.

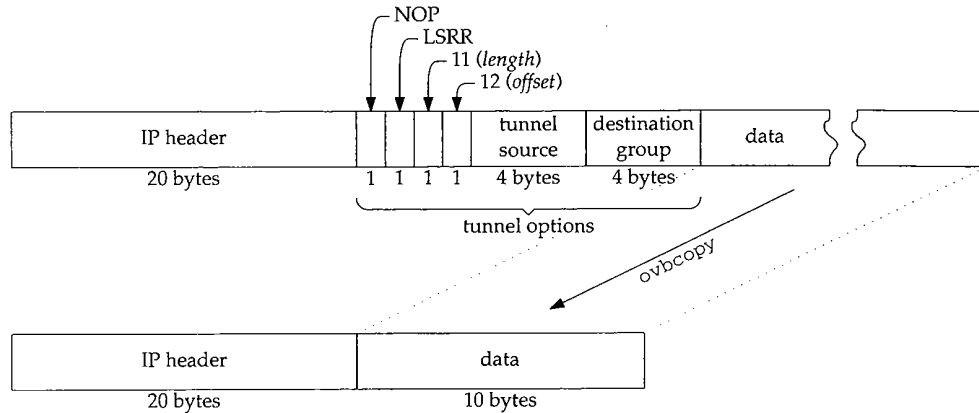


Figure 14.38 Multicast tunnel options.

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.

route.c

The next part of `ip_mforward`, shown in Figure 14.39, discards the datagram if it is ineligible for forwarding.

```

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*

580

Figure 14.39 `ip_mforward` function: forwarding eligibility checks.

#### 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 `mrouterd` 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.

593-

#### 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 `mrouterd`.

**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.

```

593      /* ip_mroute.c
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              grplst_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

```

Figure 14.40 `ip_mforward` function: forwarding.

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., `grplst_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`.

```

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
627     mb_copy = m_copy(m, 0, M_COPYALL);
628     if (mb_copy == NULL)
629         return;
630
631     imo = &simo;
632     imo->imo_multicast_ifp = vifp->v_ifp;
633     imo->imo_multicast_ttl = ip->ip_ttl - 1;
634     imo->imo_multicast_loop = 1;
635
636     error = ip_output(mb_copy, NULL, NULL, IP_FORWARDING, imo);
637 }

```

*ip\_mroute.c*

Figure 14.41 `phyint_send` function.

`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.

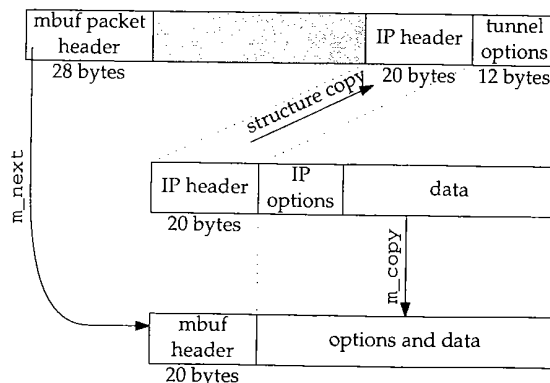


Figure 14.42 Inserting tunnel options.



**tunnel\_send Function**

1) 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.

2.c

```

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_freem(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;

```

e.c

to

re-

fi-

Figure 14.43 `tunnel_send` function: verify and allocate new header.

**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.

```

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;                /* LSRR 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

Figure 14.44 `tunnel_send` function: construct headers and send.

**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_hl` 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.

**14.9 Cleanup: ip\_mrouter\_done Function**

When mrouterd shuts down, it issues the DVMRP\_DONE command, which is handled by the ip\_mrouter\_done function shown in Figure 14.45.

```

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 (viftable[vifi].v_lcl_addr.s_addr != 0 &&
176             !(viftable[vifi].v_flags & VIFF_TUNNEL)) {
177             if (viftable[vifi].v_lcl_grps)
178                 free(viftable[vifi].v_lcl_grps, M_MRTABLE);
179             satoxin(&ifr.ifr_addr)->sin_family = AF_INET;
180             satoxin(&ifr.ifr_addr)->sin_addr.s_addr = INADDR_ANY;
181             ifp = viftable[vifi].v_ifp;
182             (*ifp->if_ioctl) (ifp, SIOCDELMULTI, (caddr_t) & ifr);
183         }
184     }
185     bzero((caddr_t) viftable, sizeof(viftable));
186     numvifs = 0;
187     /*
188      * Free any multicast route entries.
189      */
190     for (i = 0; i < MRTHASHSIZ; i++)
191         if (mrttable[i])
192             free(mrttable[i], M_MRTABLE);
193     bzero((caddr_t) mrttable, sizeof(mrttable));
194     cached_mrt = NULL;
195     ip_mrouter = NULL;
196     splx(s);
197     return (0);
198 }

```

ip\_mroute.c

Figure 14.45 ip\_mrouter\_done function: DVMRP\_DONE command.

- 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 `viftable` 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_mrout` 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.

## 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 `mrouterd`, 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?
- 14.2 Why is the update to the group membership cache in Figure 14.23 protected by `splnet` and `splx`?
- 14.3 What happens when `SIOCDELMULTI` is issued for an interface that has explicitly joined a multicast group with the `IP_ADD_MEMBERSHIP` option?
- 14.4 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.

## 15.1 Intr

This  
abst  
face  
cuss  
from

from  
crea

netv  
grat  
acce  
thro  
wri  
wor

not  
tion  
gran  
prot

# 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.

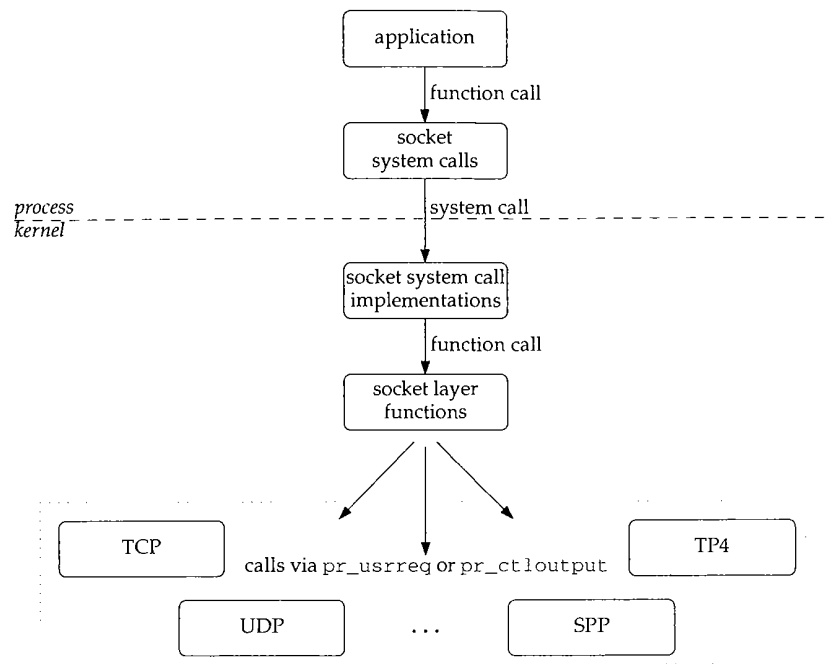


Figure 15.1 The socket layer converts generic requests to specific protocol operations.

### 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.

## 15.2 Code Introduction

The three files listed in Figure 15.2 are described in this chapter.

### Global Variables

The two global variables covered in this chapter are described in Figure 15.3.

File	Description
sys/socketvar.h	socket structure definitions
kern/uipc_syscalls.c	system call implementation
kern/uipc_socket.c	socket-layer functions

Figure 15.2 Files discussed in this chapter.

Variable	Datatype	Description
socketops	struct fileops	socket implementation of I/O system calls
sysent	struct sysent[]	array of system call entries

Figure 15.3 Global variable introduced in this chapter.

## 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.

so_options	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_KEEPAIVE</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	

Figure 15.4 `so_options` values.

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.

```

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_q0len;         /* 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_mbcnt;     /* chars of mbufs used */
76         u_long  sb_mbcmax;    /* 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 };

```

Figure 15.5 struct socket definition.



- 44 `so_linger` is the time in clock ticks that a socket waits for data to drain while closing a connection (Section 15.15).
- 45 `so_state` represents the internal state and additional characteristics of the socket. Figure 15.6 lists the possible values for `so_state`.

<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>	<ul style="list-style-type: none"> <li>•</li> <li>•</li> <li>•</li> <li>•</li> <li>•</li> <li>•</li> <li>•</li> <li>•</li> <li>•</li> </ul>	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 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

Figure 15.6 `so_state` values.

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.

`so_pcb` never points to a `tcpcb` structure directly; see Figure 22.1.

Protocol	Control block	Reference
UDP	<code>struct inpcb</code>	Section 22.3
TCP	<code>struct inpcb</code> <code>struct tcpcb</code>	Section 22.3 Section 24.5
ICMP, IGMP, raw IP	<code>struct inpcb</code>	Section 22.3
Route	<code>struct rawcb</code>	Section 20.3

Figure 15.7 Protocol control blocks.

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.

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 process 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

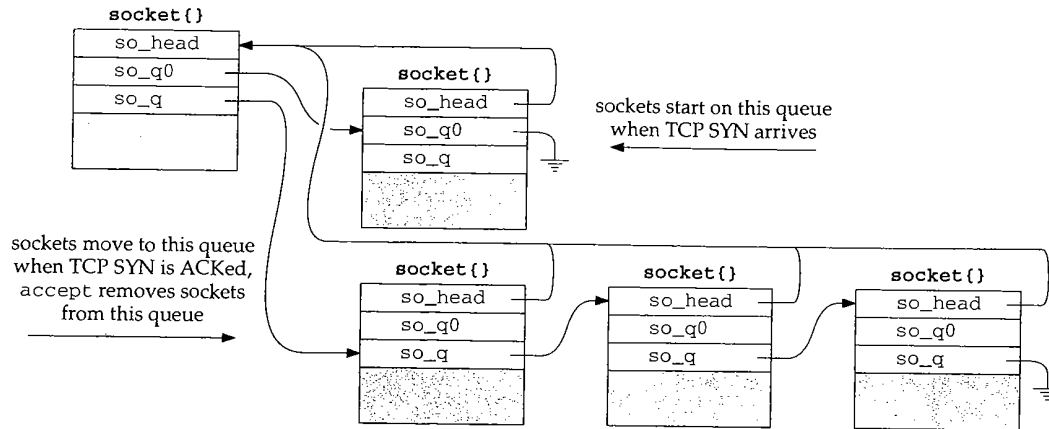


Figure 15.8 Socket connection queues.

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.

## 15.4 System Calls

A process interacts with the kernel through a collection of well-defined functions called *system calls*. Before showing the system calls that support networking, we discuss the system call mechanism itself.

The transfer of execution from a process to the protected environment of the kernel is machine- and implementation-dependent. In the discussion that follows, we use the 386 implementation of Net/3 to illustrate implementation specific operations.

In BSD kernels, each system call is numbered and the hardware is configured to transfer control to a single kernel function when the process executes a system call. The particular system call is identified as an integer argument to the function. In the 386 implementation, `syscall` is that function. Using the system call number, `syscall` indexes a table to locate the `sysent` structure for the requested system call. Each entry in the table is a `sysent` structure:

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 close 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>

Figure 15.9 Socket macros and functions.

```

struct sysent {
    int sy_narg;           /* number of arguments */
    int (*sy_call) ();    /* implementing function */
};                        /* system call table entry */

```

Here are several entries from the `sysent` array, which is defined in `kern/init_sysent.c`.

```

struct sysent sysent[] = {
    /* ... */
    { 3, recvmsg },       /* 27 = recvmsg */
    { 3, sendmsg },       /* 28 = sendmsg */
    { 6, recvfrom },      /* 29 = recvfrom */
    { 3, accept },        /* 30 = accept */
    { 3, getpeername },   /* 31 = getpeername */
    { 3, getsockname },   /* 32 = getsockname */
    /* ... */
}

```

For example, the `recvmsg` system call is the 27th entry in the system call table, has three arguments, and is implemented by the `recvmsg` function in the kernel.

`syscall` copies the arguments from the calling process into the kernel and allocates an array to hold the results of the system call, which `syscall` returns to the process when the system call completes. `syscall` dispatches control to the kernel function associated with the system call. In the 386 implementation, this call looks like:

```

struct sysent *callp;
error = (*callp->sy_call)(p, args, rval);

```

where `callp` is a pointer to the relevant `sysent` structure, `p` is a pointer to the process table entry for the process that made the system call, `args` represents the arguments to the system call as an array of 32-bit words, and `rval` is an array of two 32-bit words to hold the return value of the system call. When we use the term *system call*, we mean the function within the kernel called by `syscall`, not the function within the process called by the application.

`syscall` expects the system call function (i.e., what `sy_call` points to) to return 0 if no errors occurred and a nonzero error code otherwise. If no error occurs, the kernel passes the values in `rval` back to the process as the return value of the system call (the one made by the application). If an error occurs, `syscall` ignores the values in `rval` and returns the error code to the process in a machine-dependent way so that the error is made available to the process in the external variable `errno`. The function called by the application returns -1 or a null pointer to indicate that `errno` should be examined.

The 386 implementation sets the carry bit to indicate that the value returned by `syscall` is an error code. The system call stub in the process stores the code in `errno` and returns -1 or a null pointer to the application. If the carry bit is not set, the value returned by `syscall` is returned by the stub.

To summarize, a function implementing a system call "returns" two values: one for the `syscall` function, and a second (found in `rval`) that `syscall` returns to the calling process when no error occurs.

**Example**

The prototype for the `socket` system call is:

```
int socket(int domain, int type, int protocol);
```

The prototype for the kernel function that implements the system call is

```
struct socket_args {
    int domain;
    int type;
    int protocol;
};
socket(struct proc *p, struct socket_args *uap, int *retval);
```

When an application calls `socket`, the process passes three separate integers to the kernel with the system call mechanism. `syscall` copies the arguments into an array of 32-bit values and passes a pointer to the array as the second argument to the kernel version of `socket`. The kernel version of `socket` treats the second argument as a pointer to an `socket_args` structure. Figure 15.10 illustrates this arrangement.

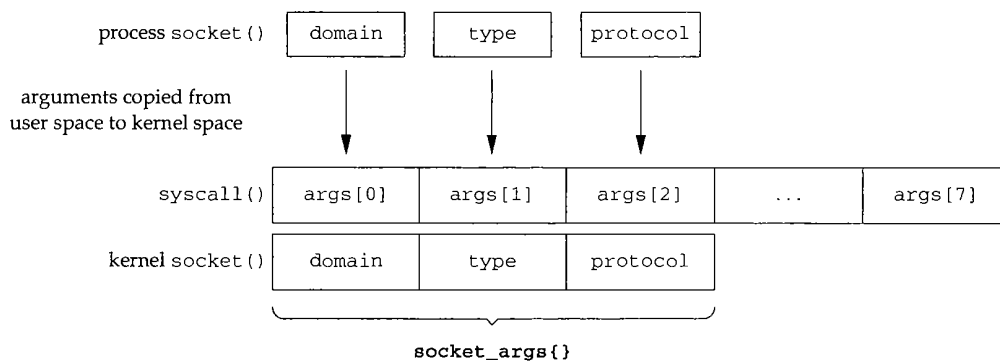


Figure 15.10 socket argument processing.

As illustrated by `socket`, each kernel function that implements a system call declares `args` not as a pointer to an array of 32-bit words, but as a pointer to a structure specific to the system call.

The implicit cast is legal only in traditional K&R C or in ANSI C when a prototype is not in effect. If a prototype is in effect, the compiler generates a warning.

`syscall` prepares the return value of 0 before executing the kernel system call function. If no error occurs, the system call function can return without clearing `*retval` and `syscall` returns 0 to the process.

**System Call Summary**

Figure 15.11 summarizes the system calls relevant to 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	prepare a socket to accept incoming connections
	accept	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
	writew	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	

Figure 15.11 Networking system calls in Net/3.

We present the setup, server, client, and termination calls in this chapter. The input and output system calls are discussed in Chapter 16 and the administrative calls in Chapter 17.

Figure 15.12 shows the sequence in which an application might use the calls. The I/O system calls in the large box can be called in any order. This is not a complete state diagram as some valid transitions are not included; just the most common ones are shown.

## 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].

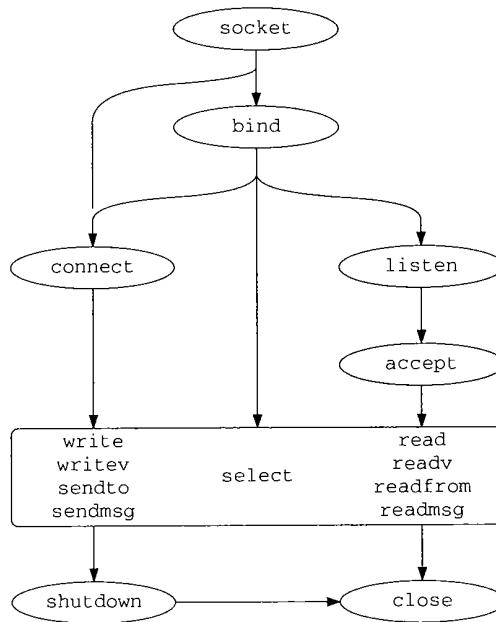


Figure 15.12 Network system call flowchart.

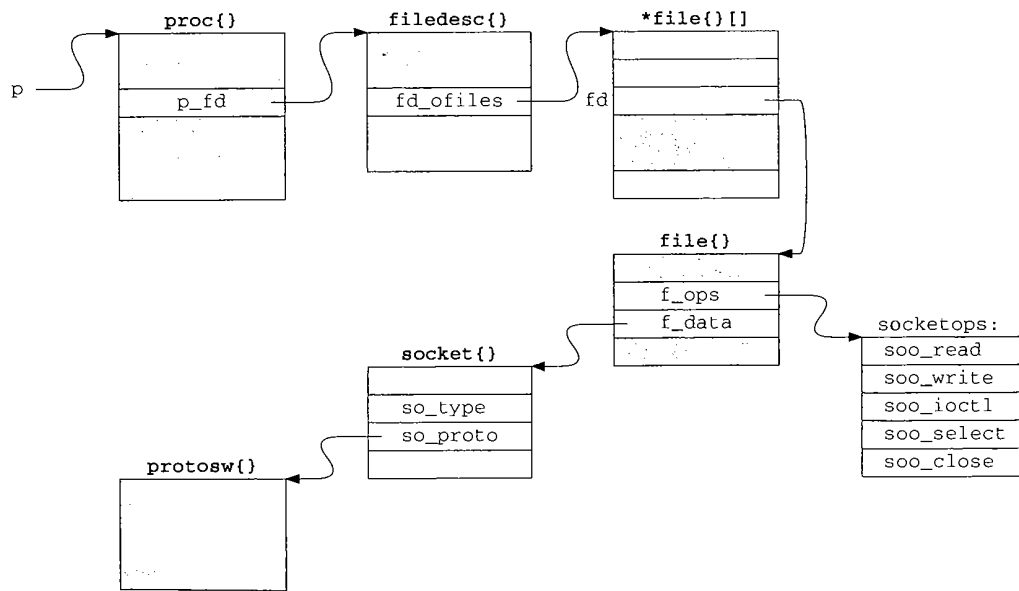


Figure 15.13 Process, file, and socket structures.

1



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.

## 15.6 socket System Call

The *socket* system call creates a new socket and associates it with a protocol as specified by the *domain*, *type*, and *protocol* arguments specified by the process. The function (shown in Figure 15.14) allocates a new descriptor, which identifies the socket in future system calls, and returns the descriptor to the process.

42-55 Before each system call a structure is defined to describe the arguments passed from the process to the kernel. In this case, the arguments are passed within a *socket\_args* structure. All the socket-layer system calls have three arguments: *p*, a pointer to the *proc* structure for the calling process; *uap*, a pointer to a structure containing the arguments passed by the process to the system call; and *retval*, a value-result argument that points to the return value for the system call. Normally, we ignore the *p* and *retval* arguments and refer to the contents of the structure pointed to by *uap* as the arguments to the system call.

56-60 *falloc* allocates a new file structure and slot in the *fd\_ofiles* array (Figure 15.13). *fp* points to the new structure and *fd* is the index of the structure in the *fd\_ofiles* array. *socket* enables the file structure for read and write access and marks it as a socket. *socketops*, a global *fileops* structure shared by all sockets, is attached to the file structure by *f\_ops*. The *socketops* variable is initialized at compile time as shown in Figure 15.15.

60-69 *socreate* allocates and initializes a *socket* structure. If *socreate* fails, the error code is posted in *error*, the file structure is released, and the descriptor slot cleared. If *socreate* succeeds, *f\_data* is set to point to the *socket* structure and establishes

ops:
read
write
ioctl
select
close

```

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_fdp;
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 = DTYPE_SOCKET;
60     fp->f_ops = &socketops;
61     if (error = socreate(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

uipc\_syscalls.c

Figure 15.14 socket system call.

Member	Value
fo_read	soo_read
fo_write	soo_write
fo_ioctl	soo_ioctl
fo_select	soo_select
fo_close	soo_close

Figure 15.15 socketops: the global fileops structure for sockets.

the association between the descriptor and the socket. `fd` is returned to the process through `*retval`. `socket` returns 0 or the error code returned by `socreate`.

### socreate Function

Most socket system calls are divided into at least two functions, in the same way that `socket` and `socreate` are. The first function retrieves from the process all the data

alls.c

required, calls the second `soxxx` function to do the work, and then returns any results to the process. This split is so that the second function can be called directly by kernel-based network protocols, such as NFS. `screate` is shown in Figure 15.16.

```

43 screate(dom, aso, type, proto)
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     if (proto)
54         prp = pffindproto(dom, proto, type);
55     else
56         prp = pffindtype(dom, type);
57     if (prp == 0 || prp->pr_usrreq == 0)
58         return (EPROTONOSUPPORT);
59     if (prp->pr_type != type)
60         return (EPROTOTYPE);
61     MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_WAIT);
62     bzero((caddr_t) so, sizeof(*so));
63     so->so_type = type;
64     if (p->p_ucred->cr_uid == 0)
65         so->so_state = SS_PRIV;
66     so->so_proto = prp;
67     error =
68         (*prp->pr_usrreq) (so, PRU_ATTACH,
69             (struct mbuf *) 0, (struct mbuf *) proto, (struct mbuf *) 0);
70     if (error) {
71         so->so_state |= SS_NOFDREF;
72         sofree(so);
73         return (error);
74     }
75     *aso = so;
76     return (0);
77 }

```

ills.c

uipc\_socket.c

Figure 15.16 `screate` function.

43-52 The four arguments to `screate` are: `dom`, the requested protocol domain (e.g., `PF_INET`); `aso`, in which a pointer to a new socket structure is returned; `type`, the requested socket 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 requested by the process. If `proto` is 0, `pffindtype` looks for a protocol within the specified domain with the semantics specified by `type`. Both functions return a pointer to a `protosw` structure of the matching protocol or a null pointer (Section 7.6).

ress

hat  
ata

**Allocate and initialize socket structure**

61-66 `screate` allocates a new socket structure, fills it with 0s, records the type, and, if the calling process has superuser privileges, turns on `SS_PRIV` in the socket structure.

**PRU\_ATTACH request**

67-69 The first example of the protocol-independent socket layer making a protocol-specific request appears in `screate`. Recall from Section 7.4 and Figure 15.13 that `so->so_proto->pr_usrreq` is a pointer to the user-request function of the protocol associated with socket `so`. Every protocol provides this function in order to handle communication requests from the socket layer. The prototype for the function is:

```
int pr_usrreq(struct socket *so, int req, struct mbuf *m0, *m1, *m2);
```

The first argument, `so`, is a pointer to the relevant socket and `req` is a constant identifying the particular request. The next three arguments (`m0`, `m1`, and `m2`) are different for each request. They are always passed as pointers to `mbuf` structures, even if they have another type. Casts are used when necessary to avoid warnings from the compiler.

Figure 15.17 shows the requests available through the `pr_usrreq` function. The semantics of each request depend on the particular protocol servicing the request.

Request	Arguments			Description
	<i>m0</i>	<i>m1</i>	<i>m2</i>	
<code>PRU_ABORT</code>				abort any existing connection
<code>PRU_ACCEPT</code>		<i>address</i>		wait for and accept a connection
<code>PRU_ATTACH</code>		<i>protocol</i>		a new socket has been created
<code>PRU_BIND</code>		<i>address</i>		bind the address to the socket
<code>PRU_CONNECT</code>		<i>address</i>		establish association or connection to address
<code>PRU_CONNECT2</code>		<i>socket2</i>		connect two sockets together
<code>PRU_DETACH</code>				socket is being closed
<code>PRU_DISCONNECT</code>				break association between socket and foreign address
<code>PRU_LISTEN</code>				begin listening for connections
<code>PRU_PEERADDR</code>		<i>buffer</i>		return foreign address associated with socket
<code>PRU_RCVD</code>		<i>flags</i>		process has accepted some data
<code>PRU_RCVOOB</code>	<i>buffer</i>	<i>flags</i>		receive OOB data
<code>PRU_SEND</code>	<i>data</i>	<i>address</i>	<i>control</i>	send regular data
<code>PRU_SENDOOB</code>	<i>data</i>	<i>address</i>	<i>control</i>	send OOB data
<code>PRU_SHUTDOWN</code>				end communication with foreign address
<code>PRU_SOCKADDR</code>		<i>buffer</i>		return local address associated with socket

Figure 15.17 `pr_usrreq` requests.

`PRU_CONNECT2` is supported only within the Unix domain, where it connects two local sockets to each other. Unix pipes are implemented in this way.

**Cleanup and return**

70-77 Returning to `screate`, the function attaches the protocol switch table to the new socket and issues the `PRU_ATTACH` request to notify the protocol of the new end point. This request causes most protocols, including TCP and UDP, to allocate and initialize any structures required to support the new end point.

## Superuser Privileges

Figure 15.18 summarizes the networking operations that require superuser access.

Function	Superuser		Description	Reference
	Process	Socket		
<code>in_control</code>		•	interface address, netmask, and destination address assignment	Figure 6.14
<code>in_control</code>		•	broadcast address assignment	Figure 6.22
<code>in_pcbbind</code>	•		binding to an Internet port less than 1024	Figure 22.22
<code>ifioctl</code>	•		interface configuration changes	Figure 4.29
<code>ifioctl</code>	•		multicast address configuration (see text)	Figure 12.11
<code>rip_usrreq</code>	•		creating an ICMP, IGMP, or raw IP socket	Figure 32.10
<code>slopen</code>	•		associating a SLIP device with a tty device	Figure 5.9

Figure 15.18 Superuser privileges in Net/3.

The multicast `ioctl` commands (`SIOCADMULTI` and `SIOCDELMULTI`) are accessible to non-superuser processes when they are invoked indirectly by the `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP` socket options (Sections 12.11 and 12.12).

In Figure 15.18, the "Process" column identifies requests that must be made by a superuser process, and the "Socket" column identifies requests that must be issued on a socket *created* by a superuser process (i.e., the process does not need superuser privileges if it has access to the socket, Exercise 15.1). In Net/3, the `suser` function determines if the calling process has superuser privileges, and the `SS_PRIV` flag determines if the socket was created by a superuser process.

Since `rip_usrreq` tests `SS_PRIV` immediately after creating the socket with `screate`, we show this function as accessible only from a superuser process.

## 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.

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.

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

```

754 getsock(fdp, fdes, fpp)
755 struct filedesc *fdp;
756 int    fdes;
757 struct file **fpp;
758 {
759     struct file *fp;
760     if ((unsigned) fdes >= fdp->fd_nfiles ||
761         (fp = fdp->fd_ofiles[fdes]) == NULL)
762         return (EBADF);
763     if (fp->f_type != DTYPE_SOCKET)
764         return (ENOTSOCK);
765     *fpp = fp;
766     return (0);
767 }

```

Figure 15.19 getsock function.

```

768 sockargs(mp, buf, buflen, type)
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 }

```

Figure 15.20 sockargs function.

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.

## 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`.

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.

s.c

he  
ed  
ss

```

-----uipc_syscalls.c
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 }
-----uipc_syscalls.c

```

Figure 15.21 bind function.

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.

```

-----uipc_socket.c
78 sobind(so, nam)
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 }
-----uipc_socket.c

```

Figure 15.22 sobind function.



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.

## 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).

```

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 }

```

*uipc\_syscalls.c*

*uipc\_syscalls.c*

Figure 15.23 listen system call.

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.

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.

```

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*

*uipc\_socket.c*

Figure 15.24 solisten function.

## 15.10 tsleep and wakeup Functions

When a process executing within the kernel cannot proceed because a kernel resource is unavailable, it waits for the resource by calling `tsleep`, which has the following prototype:

```
int tsleep(caddr_t chan, int pri, char *mesg, int timeo);
```

The first argument to `tsleep`, `chan`, is called the *wait channel*. It identifies the particular resource or event such as an incoming network connection, for which the process is waiting. Many processes can be sleeping on a single wait channel. When the resource becomes available or when the event occurs, the kernel calls `wakeup` with the wait channel as the single argument. The prototype for `wakeup` is:

```
void wakeup(caddr_t chan);
```

All processes waiting for the channel are awakened and set to the run state. The kernel arranges for `tsleep` to return when each of the processes resumes execution.

The `pri` argument specifies the priority of the process when it is awakened, as well as several optional control flags for `tsleep`. By setting the `PCATCH` flag in `pri`, `tsleep` also returns when a signal arrives. `mesg` is a string identifying the call to `tsleep` and is included in debugging messages and in `ps` output. `timeo` sets an upper bound on the sleep period and is measured in clock ticks.

Figure 15.25 summarizes the return values from `tsleep`.

A process never sees the `ERESTART` error because it is handled by the `syscall` function and never returned to a process.

tsleep()	Description
0	The process was awakened by a matching call to <code>wakeup</code> .
EWOLDBLOCK	The process was awakened after sleeping for <code>timeo</code> clock ticks and before the matching call to <code>wakeup</code> .
ERESTART	A signal was handled by the process during the sleep and the pending system call should be restarted.
EINTR	A signal was handled by the process during the sleep and the pending system call should fail.

Figure 15.25 `tsleep` return values.

Because all processes sleeping on a wait channel are awakened by `wakeup`, we always see a call to `tsleep` within a tight loop. Every process must determine if the resource is available before proceeding because another awakened process may have claimed the resource first. If the resource is not available, the process calls `tsleep` once again.

It is unusual for multiple processes to be sleeping on a single socket, so a call to `wakeup` usually causes only one process to be awakened by the kernel.

For a more detailed discussion of the sleep and wakeup mechanism see [Leffler et al. 1989].

### Example

One use of multiple processes sleeping on the same wait channel is to have multiple server processes reading from a UDP socket. Each server calls `recvfrom` and, as long as no data is available, the calls block in `tsleep`. When a datagram arrives on the socket, the socket layer calls `wakeup` and each server is placed on the run queue. The first server to run receives the datagram while the others call `tsleep` again. In this way, incoming datagrams are distributed to multiple servers without the cost of starting a new process for each datagram. This technique can also be used to process incoming connection requests in TCP by having multiple processes call `accept` on the same socket. This technique is described in [Comer and Stevens 1993].

## 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`.

```
106 struct accept_args {
107     int     s;
108     caddr_t name;
109     int     *anamelen;
110 };
111
112 accept(p, uap, retval)
113 struct proc *p;
114 struct accept_args *uap;
115 int *retval;
116 {
117     struct file *fp;
118     struct mbuf *nam;
119     int     namelen, error, s;
120     struct socket *so;
121
122     if (uap->name && (error = copyin((caddr_t) uap->anamelen,
123                                     (caddr_t) & namelen, sizeof(namelen))))
124         return (error);
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     }
148     if (so->so_error) {
149         error = so->so_error;
150         so->so_error = 0;
151         splx(s);
152         return (error);
153     }
154     if (error = falloc(p, &fp, retval)) {
155         splx(s);
156         return (error);
157     }
158 }
```

*uipc\_syscalls.c*

```

156     { struct socket *aso = so->so_q;
157       if (soqremque(aso, 1) == 0)
158         panic("accept");
159       so = aso;
160     }

161     fp->f_type = DTYPE_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*

Figure 15.26 accept system call.

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.