

TCP/IP Illustrated, Volume 2
The Implementation

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



ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Contents

Preface	xix
Chapter 1. Introduction	1
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	
Chapter 2. Mbufs: Memory Buffers	31
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
Chapter 3.	Interface Layer	63
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
Chapter 4.	Interfaces: Ethernet	95
4.1	Introduction	95
4.2	Code Introduction	96
4.3	Ethernet Interface	98
4.4	ioctl System Call	114
4.5	Summary	125
Chapter 5.	Interfaces: SLIP and Loopback	127
5.1	Introduction	127
5.2	Code Introduction	127
5.3	SLIP Interface	128
5.4	Loopback Interface	150
5.5	Summary	153
Chapter 6.	IP Addressing	155
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

63

	Chapter 7. Domains and Protocols	185
	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 pffindproto and pffindtype Functions 196	
	7.7 pfctlinput Function 198	
	7.8 IP Initialization 199	
	7.9 sysctl System Call 201	
	7.10 Summary 204	
95	Chapter 8. IP: Internet Protocol	205
	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	
127	Chapter 9. IP Option Processing	247
	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_insertoptions Function 265	
	9.9 ip_pcbopts Function 269	
	9.10 Limitations 272	
	9.11 Summary 272	
155	Chapter 10. IP Fragmentation and Reassembly	275
	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	

Chapter 11. ICMP: Internet Control Message Protocol	301
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
Chapter 12. IP Multicasting	337
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
Chapter 13. IGMP: Internet Group Management Protocol	381
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

301

Chapter 14. IP Multicast Routing

397

- 14.1 Introduction 397
- 14.2 Code Introduction 398
- 14.3 Multicast Output Processing Revisited 399
- 14.4 mrouted 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_mrouter_done Function 433
- 14.10 Summary 434

337

Chapter 15. Socket Layer

435

- 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

381

Chapter 16. Socket I/O

475

- 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, ready, 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

Chapter 17. Socket Options	537
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	
Chapter 18. Radix Tree Routing Tables	559
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	
Chapter 19. Routing Requests and Routing Messages	601
19.1 Introduction 601	
19.2 rtalloc and rtalloc1 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_msgl 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	
Chapter 20. Routing Sockets	645
20.1 Introduction 645	
20.2 routedomain and protosw Structures 646	
20.3 Routing Control Blocks 647	

537	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	
559	Chapter 21. ARP: Address Resolution Protocol 675 21.1 Introduction 675 21.2 ARP and the Routing Table 675 21.3 Code Introduction 678 21.4 ARP Structures 681 21.5 arpwhohas 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	
601	Chapter 22. Protocol Control Blocks 713 22.1 Introduction 713 22.2 Code Introduction 715 22.3 inpcb Structure 716 22.4 in_pcbaalloc and in_pcbadetach Functions 717 22.5 Binding, Connecting, and Demultiplexing 719 22.6 in_pcblockup Function 724 22.7 in_pcbbind Function 728 22.8 in_pcbaconnect Function 735 22.9 in_pcbadisconnect Function 741 22.10 in_setsockaddr and in_setpeeraddr Functions 741 22.11 in_pcbanotify, in_rtchange, and in_losing Functions 742 22.12 Implementation Refinements 750 22.13 Summary 751	
645	Chapter 23. UDP: User Datagram Protocol 755 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
Chapter 24. TCP: Transmission Control Protocol		795
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
Chapter 25. TCP Timers		817
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
Chapter 26. TCP Output		851
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

795

817

851

	Chapter 27. TCP Functions	891
	27.1 Introduction 891	
	27.2 <i>tcp_drain</i> Function 892	
	27.3 <i>tcp_drop</i> Function 892	
	27.4 <i>tcp_close</i> Function 893	
	27.5 <i>tcp_mss</i> Function 897	
	27.6 <i>tcp_ctlinput</i> Function 904	
	27.7 <i>tcp_notify</i> Function 904	
	27.8 <i>tcp_quench</i> Function 906	
	27.9 <i>TCP_REASS</i> Macro and <i>tcp_reass</i> Function 906	
	27.10 <i>tcp_trace</i> Function 916	
	27.11 Summary 920	
	Chapter 28. TCP Input	923
	28.1 Introduction 923	
	28.2 Preliminary Processing 925	
	28.3 <i>tcp_dooptions</i> 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	
	Chapter 29. TCP Input (Continued)	967
	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 <i>tcp_pullingoutofband</i> 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	
	Chapter 30. TCP User Requests	1007
	30.1 Introduction 1007	
	30.2 <i>tcp_usrreq</i> Function 1007	
	30.3 <i>tcp_attach</i> Function 1018	
	30.4 <i>tcp_disconnect</i> Function 1019	

30.5	tcp_usrisclosed Function	1021
30.6	tcp_ctloutput Function	1022
30.7	Summary	1025
Chapter 31.	BPF: BSD Packet Filter	1027
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
Chapter 32.	Raw IP	1049
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
Epilogue		1067
Appendix A.	Solutions to Selected Exercises	1069
Appendix B.	Source Code Availability	1093
Appendix C.	RFC 1122 Compliance	1097
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
Bibliography		1125
Index		1133

1027

Preface

1049

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.

1067

1069

1093

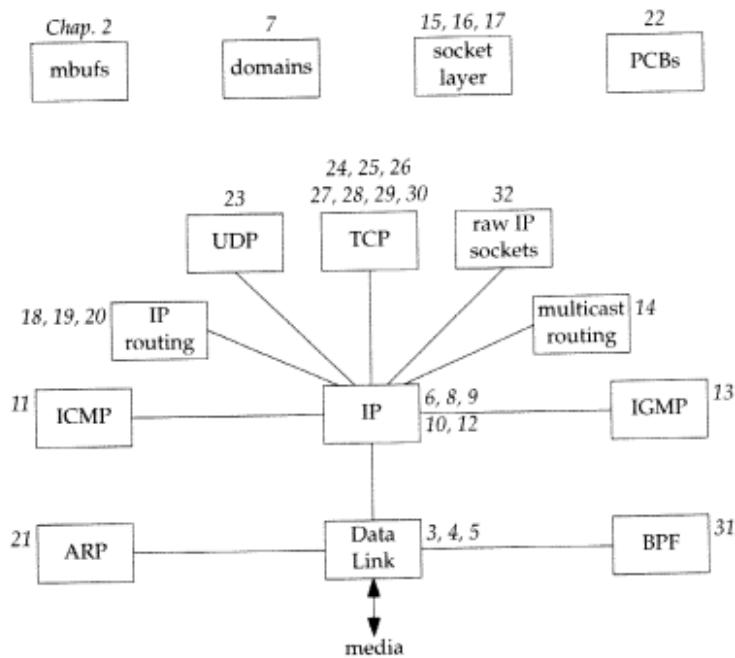
1097

1125

1133

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, Chakravardhi Ravi, Peter Salus, Doug Schmidt, Keith Sklower, Ian Lance Taylor, and G. N. Ananda Vardhana. A special thanks to the consulting editor, Brian Kernighan, for his rapid, thorough, and helpful reviews throughout the course of the project, and for his continued encouragement and support.

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

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

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

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

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

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

Gary R. Wright

<http://www.connix.com/~gwright>
Middletown, Connecticut

November 1994

W. Richard Stevens

<http://www.kohala.com/~rstevens>
Tucson, Arizona

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:

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

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:

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

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
<code>M_BCAST</code>	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

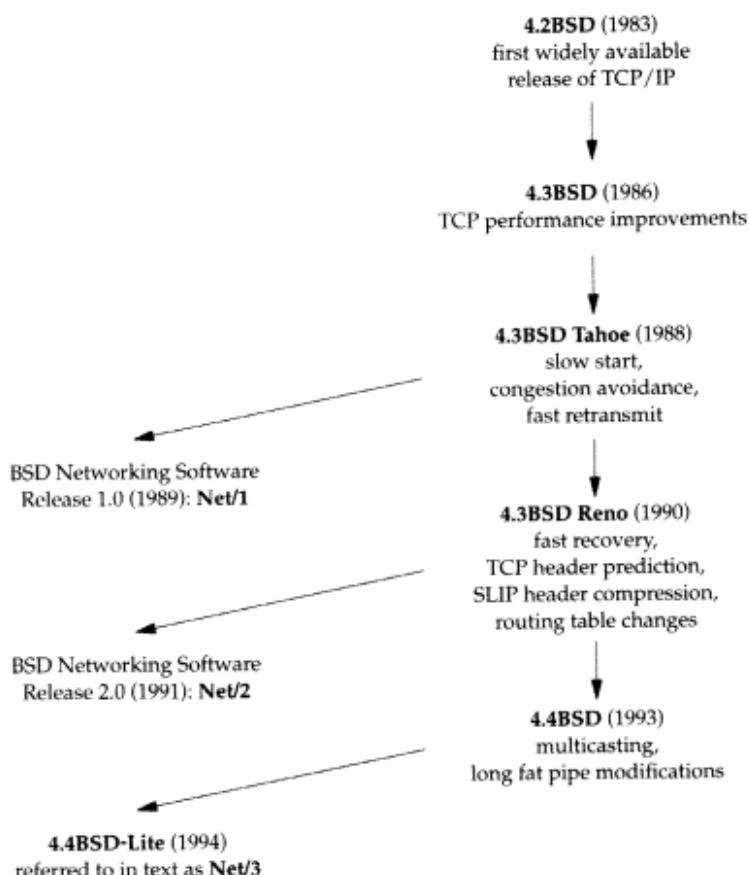


Figure 1.1 Various BSD releases with important TCP/IP features.

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

```

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 `htonl` 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, `htonl` is typically a macro that does nothing. In BSD/386, however, on the little endian 80386, `htonl` 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

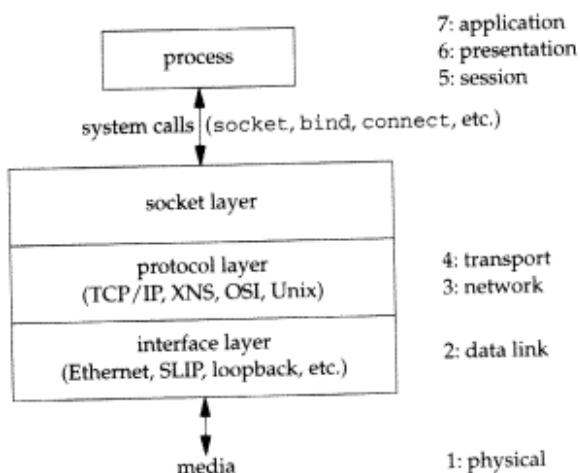


Figure 1.3 The general organization of networking code in Net/3.

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.

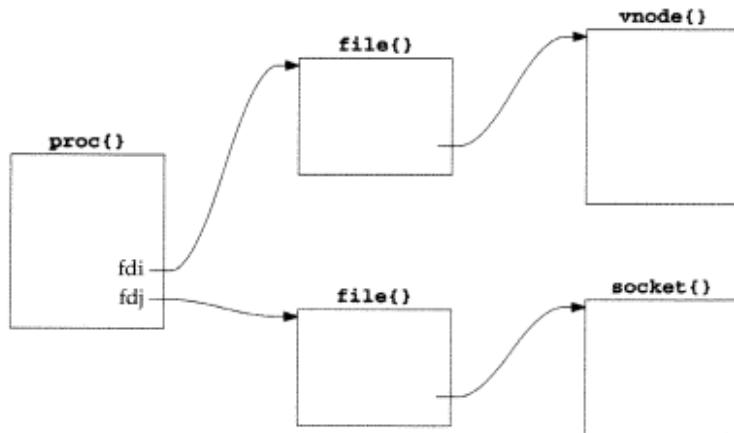


Figure 1.4 Fundamental relationship between kernel data structures starting with a descriptor.

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.

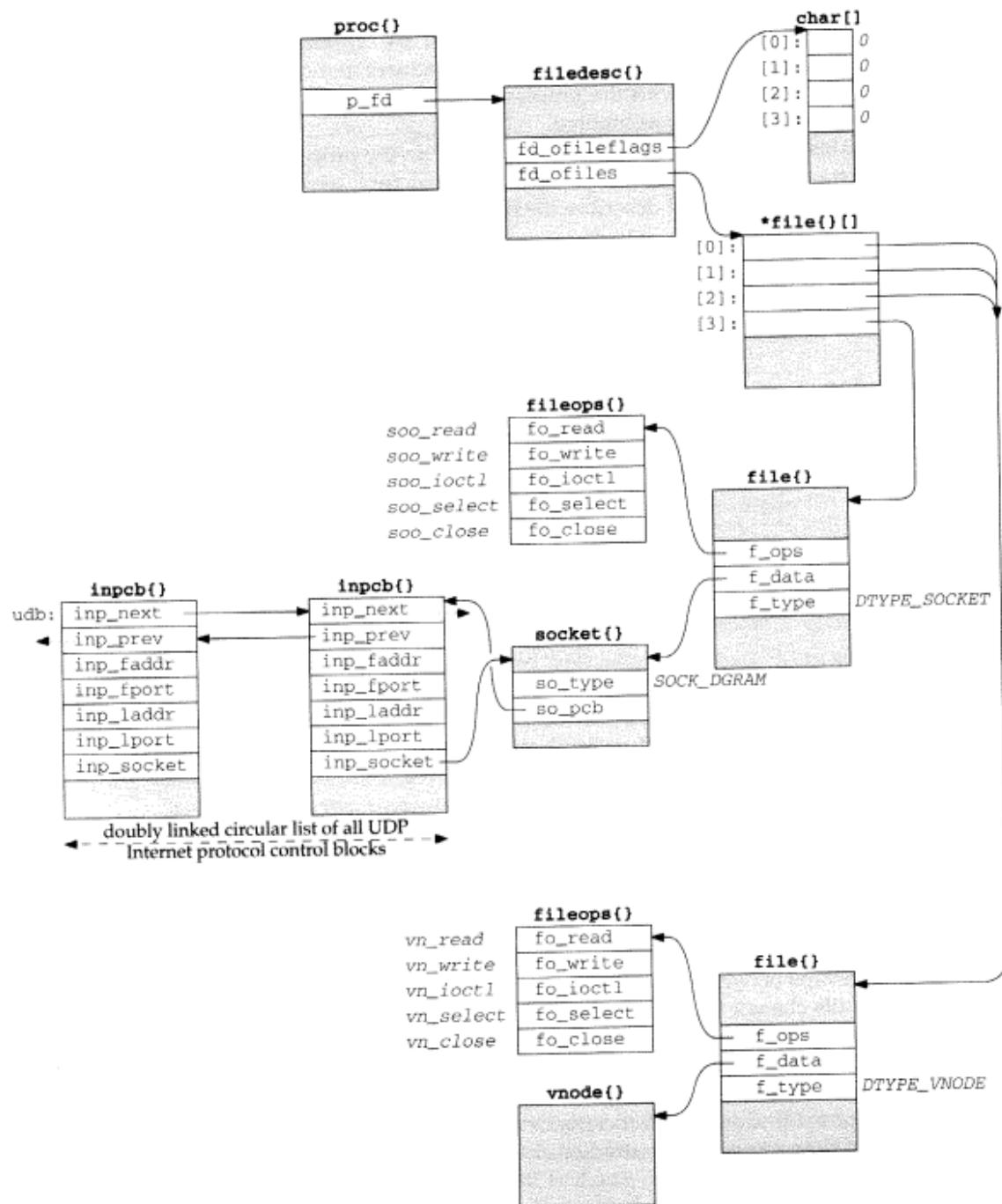


Figure 1.5 Kernel data structures after call to `socket` in example program.

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`, `writev`, `ioctl`, `select`, and `close` system calls, since these system calls work with either a socket descriptor or a nonsocket descriptor. Rather than look at the `f_type` value each time one of these system calls is invoked and then jump accordingly, the implementors chose always to jump indirectly through the corresponding entry in the `fileops` structure instead.

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

Next we come to the `socket` structure that is pointed to by the `file` structure when the descriptor type is `DTYPE_SOCKET`. In our example, the socket type (`SOCK_DGRAM` for a datagram socket) is stored in the `so_type` member. An Internet protocol control block (PCB) is also allocated: an `inpcb` structure. The `so_pcbs` 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 `edb` 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 `edb` structure, not the `inp_prev` member of that structure. The dotted arrows from `edb.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.

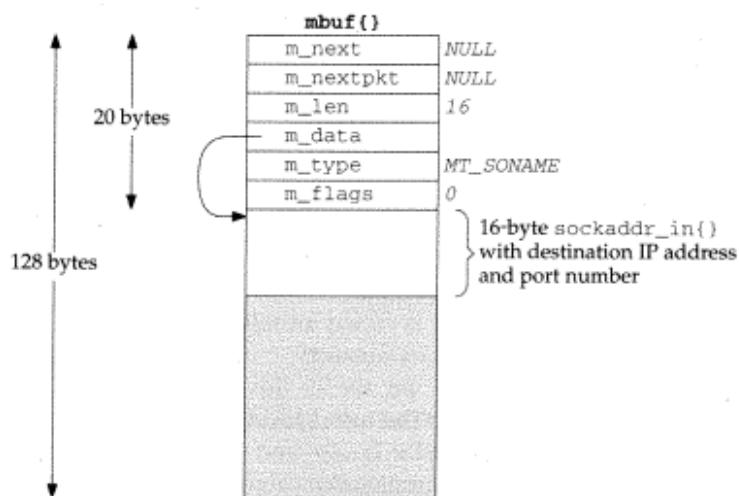


Figure 1.6 Mbuf containing destination address for `sendto`.

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 mbus. 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 mbus hold the 150 bytes of data.

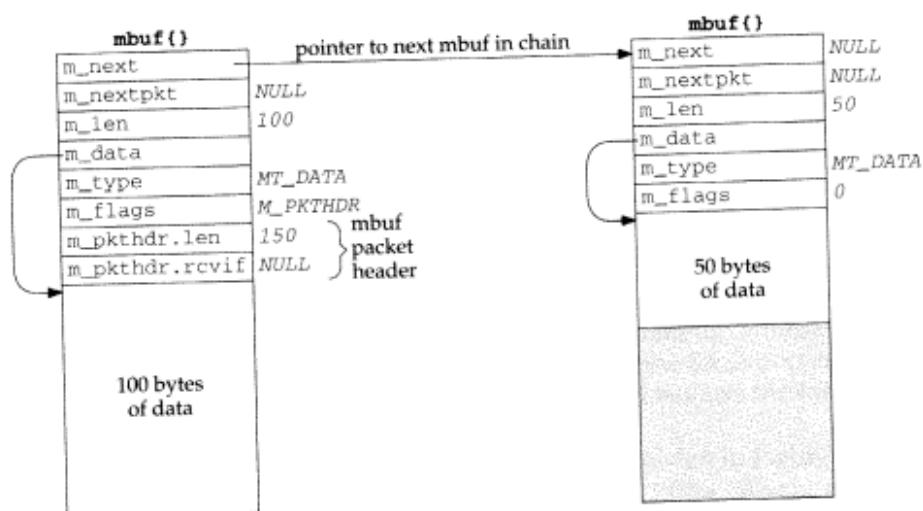


Figure 1.7 Two mbus holding 150 bytes of data.

This arrangement is called an *mbuf chain*. The `m_next` member in each mbuf links together all the mbus 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 mbus 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 mbus on the chain, two mbus 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 mbus, 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 mbus on the chain to sum their `m_len` members when the total length is needed.

Prepending IP and UDP Headers

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

The way that data is prepended to the mbuf chain in Figure 1.7 is to allocate another mbuf, make it the front of the chain, and copy the packet header from the mbuf with 100 bytes of data into the new mbuf. This gives us the three mbufs shown in Figure 1.8.

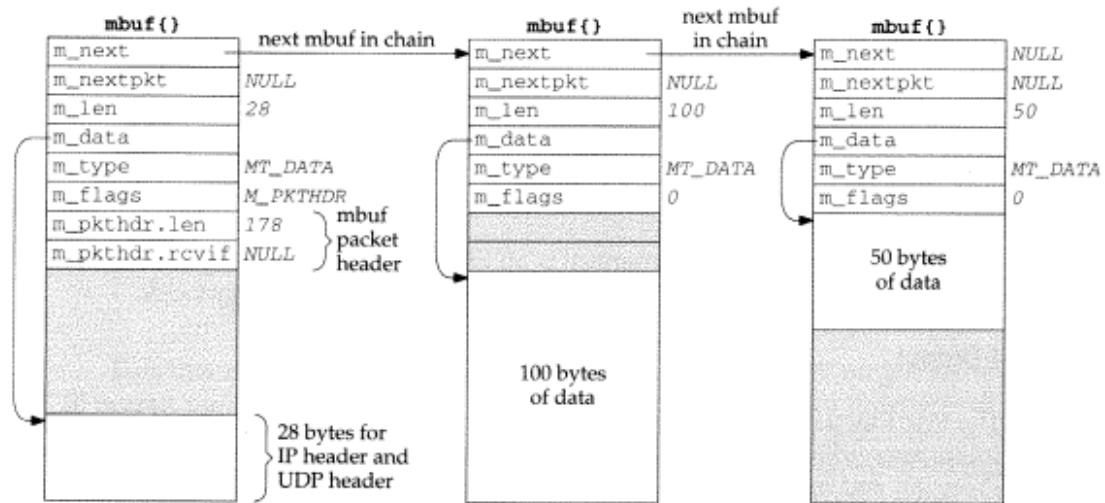


Figure 1.8 Mbuf chain from Figure 1.7 with another mbuf for IP and UDP headers prepended.

The IP header and UDP header are stored at the end of the new mbuf that becomes the head of the chain. This allows for any lower-layer protocols (e.g., the interface layer) to prepend its headers in front of the IP header if necessary, without having to copy the IP and UDP headers. The `m_data` pointer in the first mbuf points to the start of these two headers, and `m_len` is 28. Future headers that fit in the 72 bytes of unused space between the packet header and the IP header can be prepended before the IP header by adjusting the `m_data` pointer and the `m_len` accordingly. Shortly we'll see that the Ethernet header is built here in this fashion.

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

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

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

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

IP Output

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

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

Ethernet Output

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

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

The mbuf chain is then added to the end of the output queue for the interface. If the interface is not currently busy, the interface's "start output" routine is called directly. If the interface is busy, its output routine will process the new mbuf on its queue when it is finished with the buffers already on its output queue.

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

Summary of UDP Output

In Figure 1.9 we give an overview of the processing that takes place when a process calls `sendto` to transmit a single UDP datagram. The relationship of the processing that we've described to the three layers of kernel code (Figure 1.3) is also shown.

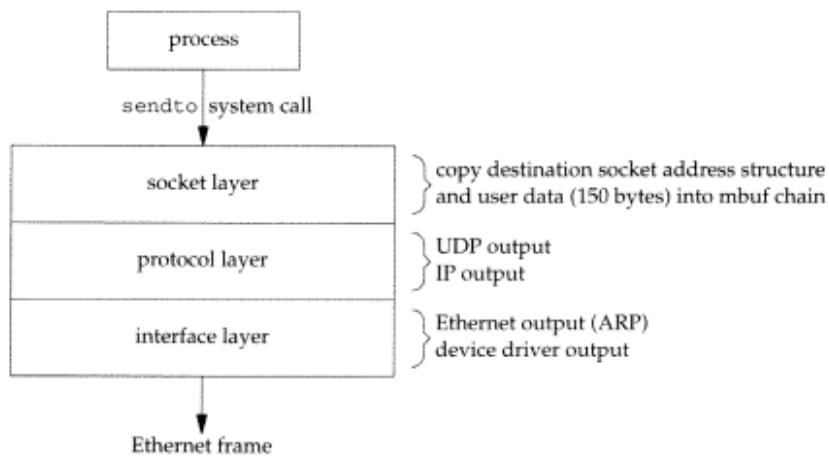


Figure 1.9 Processing performed by the three layers for simple UDP output.

Function calls pass control from the socket layer to the UDP output routine, to the IP output routine, and then to the Ethernet output routine. Each function call passes a pointer to the mbuf chain to be output. At the lowest layer, the device driver, the mbuf chain is placed on the device's output queue and the device is started, if necessary. The function calls return in reverse order of their call, and eventually the system call returns to the process. Notice that there is no queueing of the UDP data until it arrives at the device driver. The higher layers just prepend their header and pass the mbuf to the next lower layer.

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

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.

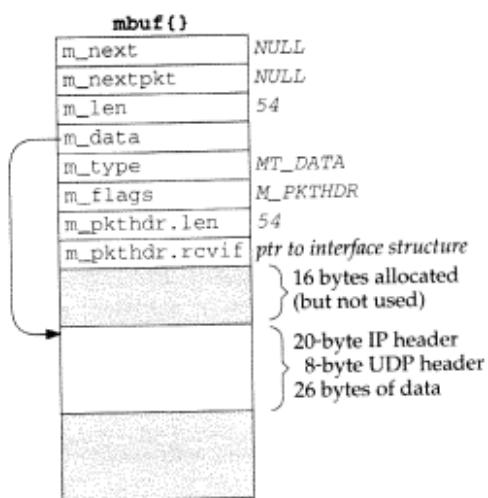


Figure 1.10 Single mbuf to hold input Ethernet data.

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 `recvmmsg` 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`).

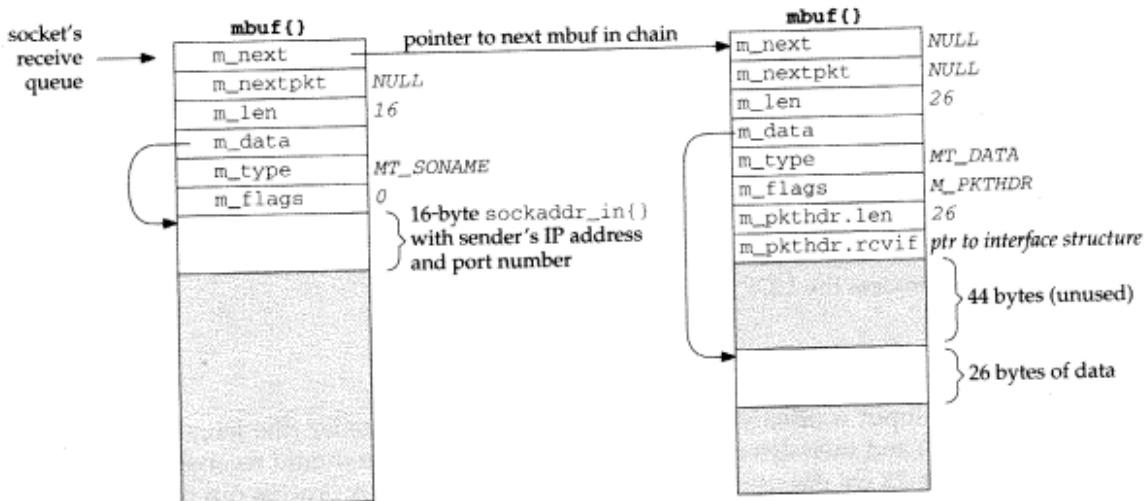


Figure 1.11 Sender's address and data.

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.

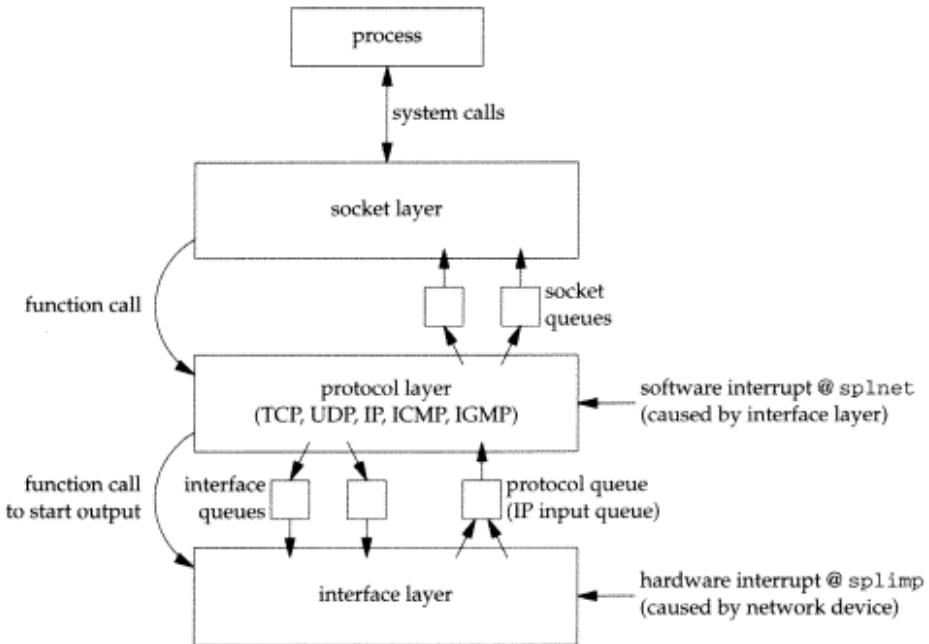


Figure 1.12 Communication between the layers for network input and output.

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

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.

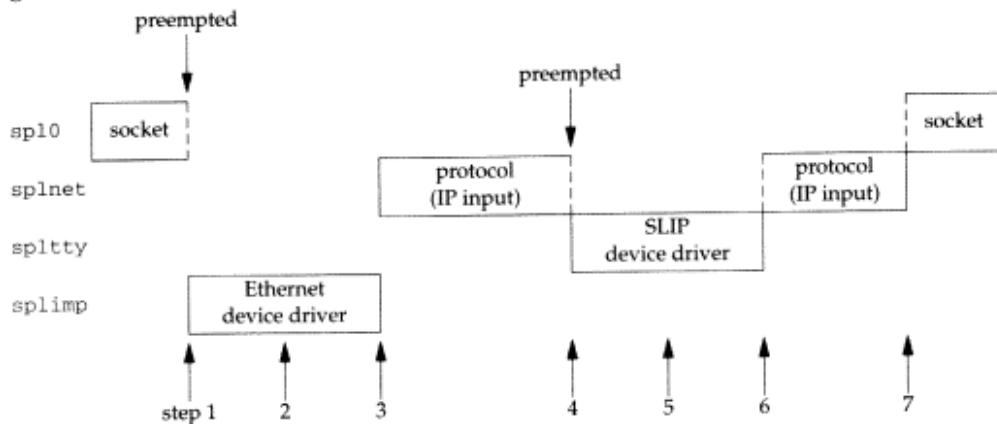


Figure 1.14 Example of priority levels and kernel processing.

1. While the socket layer is executing at `spl0`, 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.

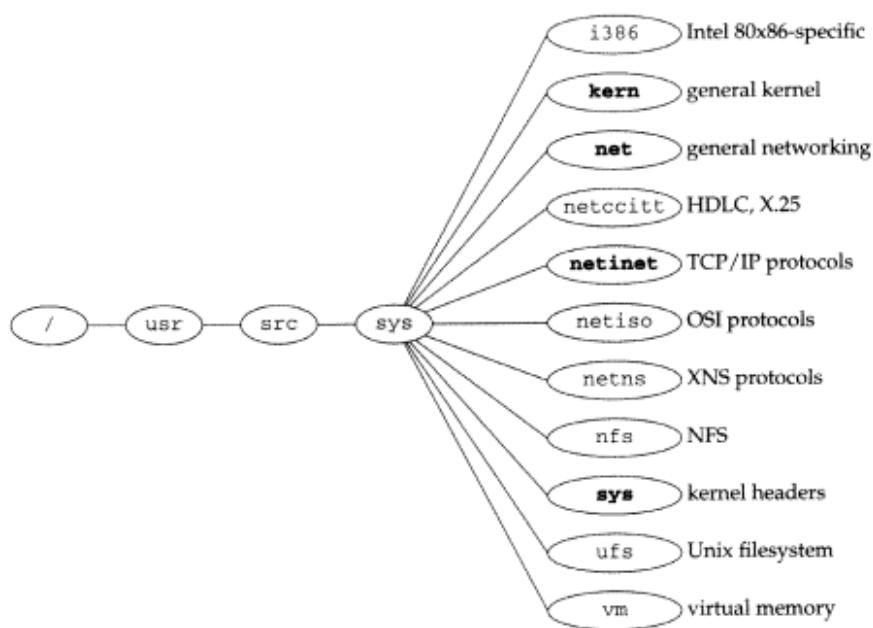


Figure 1.15 Net/3 source code organization.

- **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.

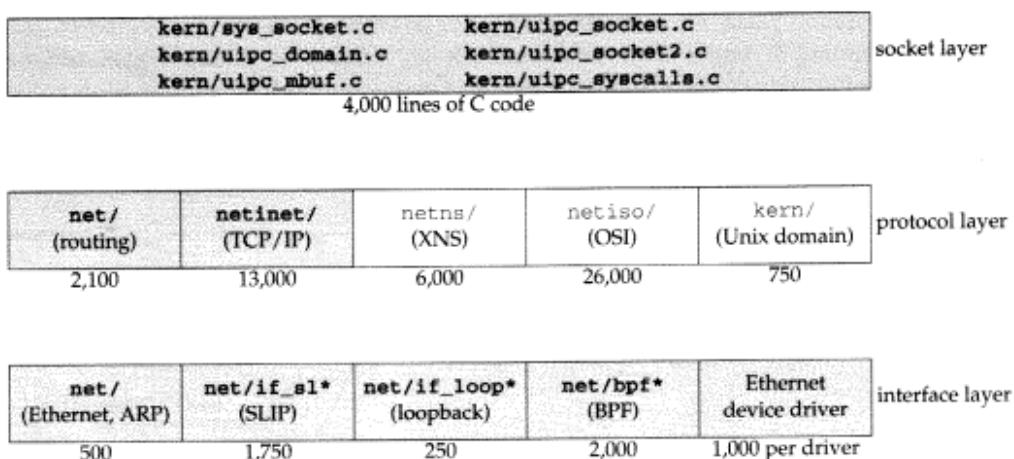


Figure 1.16 Net/3 source code organization mapped to three kernel layers.

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.

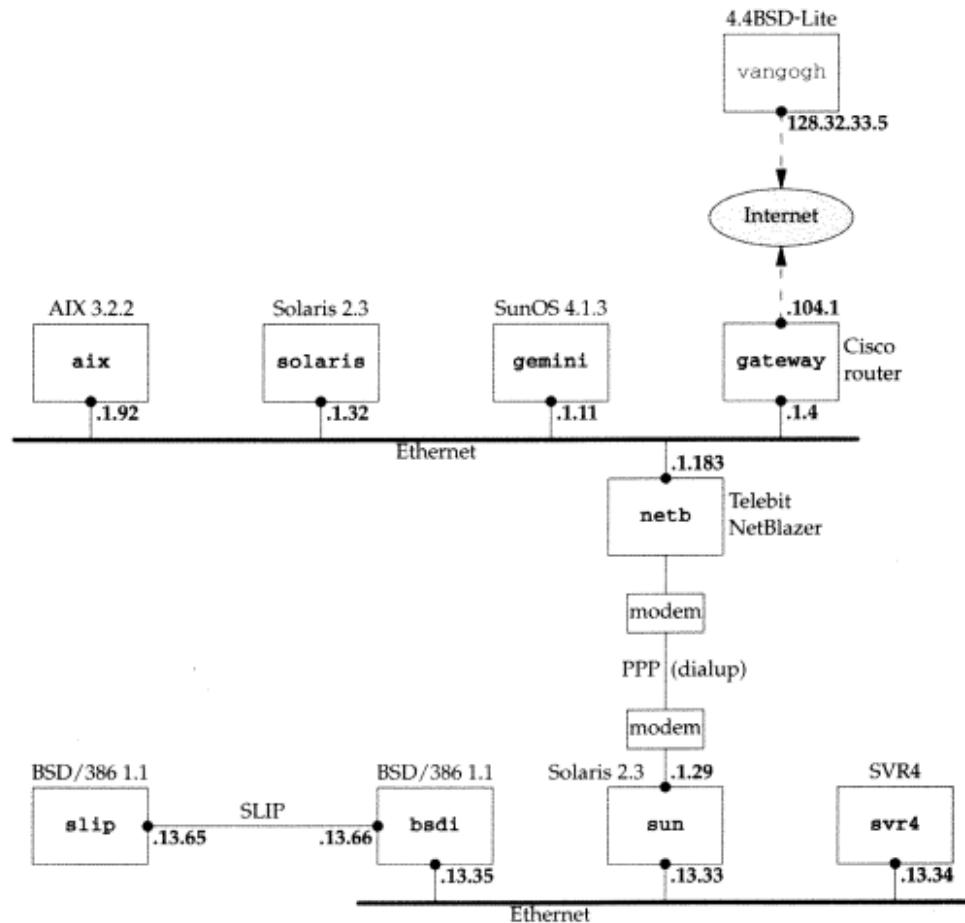


Figure 1.17 Test network used for all the examples in the text.

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 `spl` functions in almost every function that we look at.

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

Exercises

- 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_dat` array). The `m_data` pointer points somewhere in this 108-byte buffer. We show it pointing to the start of the buffer, but it can point anywhere in the buffer. The `m_len` member specifies the number of bytes of data, starting at `m_data`. Figure 1.6 was an example of this type of mbuf.

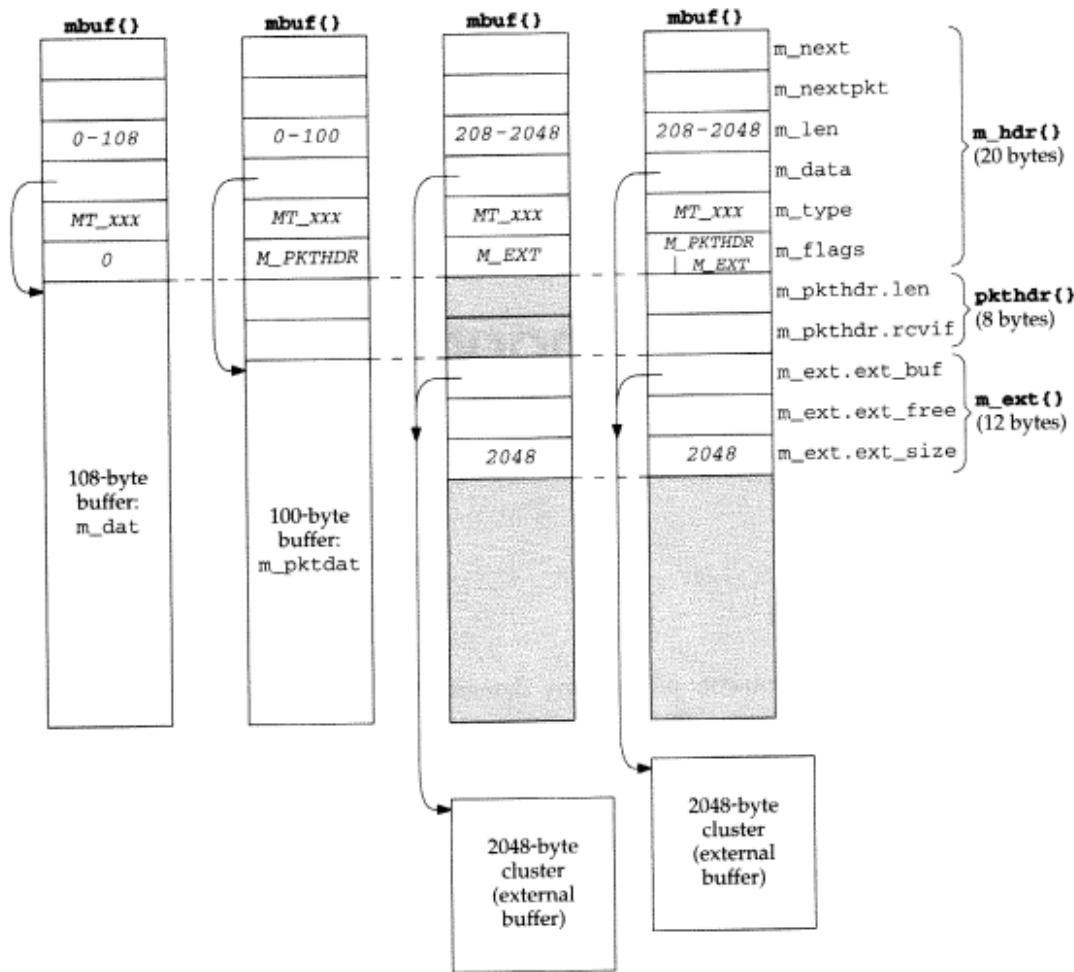


Figure 2.1 Four different types of mbufs, depending on the m_flags value.

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_pktdat array). Figure 1.10 was an example of this type of mbuf.

The m_pkthdr.len value is the total length of all the data in the mbuf chain for this packet: the sum of the m_len values for all the mbufs linked through the

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

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 mbus 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 mbus, 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 mbus 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).

beginning of
this pointer
sent.

the buffer
size of 2048
are not the
buffer. The
ly used by

set (record)

to form a
mbuf chain.
mbufs define
used—the
pointers.

tion of Fig-
e (showing
the first mbuf
containing
uf has been
e show that
cluster. We
erface out-
to the mbuf
this mbuf.

DATA, but the
This is a side
makes no dif-
ags value of

/3 mbuf, Fig-
were made in
nextpkt, and

UDP and TCP
while TCP

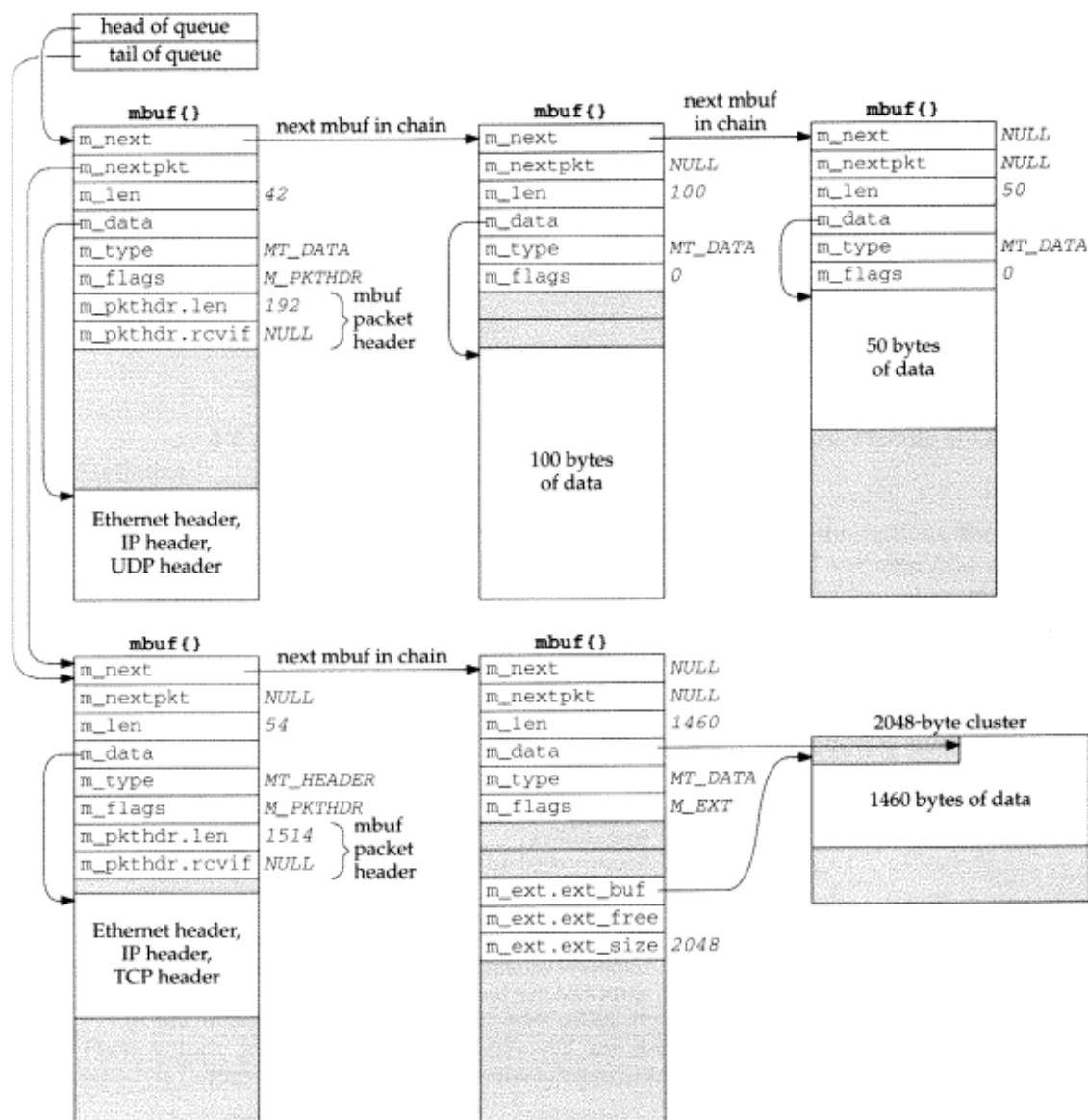


Figure 2.2 Two packets on a queue: first with 192 bytes of data and second with 1514 bytes of data.

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

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 (99×128 bytes) plus the cluster memory (34×2048 bytes) divided by 1024. The percentage in use is the mbuf memory (99×128 bytes) plus the cluster memory in use (2×2048 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_mtotypes[MT_DATA]
43 mbufs allocated to packet headers	m_mtotypes[MT_HEADER]
17 mbufs allocated to protocol control blocks	m_mtotypes[MT_PCB]
20 mbufs allocated to socket names and addresses	m_mtotypes[MT SONAME]
18 mbufs allocated to socket options	m_mtotypes[MT_SOOPTS]
2/34 mapped pages in use	(see text)
80 Kbytes allocated to network (20% in use)	(see text)
0 requests for memory denied	m_drops
0 requests for memory delayed	m_wait
0 calls to protocol drain routines	m_drain

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.

```
mbuf.h
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 *recvif;         /* 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
```

mbuf.h

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.

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

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

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, mbtypes[type], (how)); \
156     if ((m) == NULL) { \
157         (m)->m_type = (type); \
158         MBUFLOCK(mbstat.m_mtotypes[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     }

```

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 `mbtypes` converts the mbuf `MT_xxx` value into the corresponding `M_xxx` value (Figure 2.10). If the memory can be allocated, the `m_type` member is set to the argument's value.

158 The kernel structure that keeps mbuf statistics for each type of mbuf is incremented (`mbstat`). The macro `MBUFLOCK` changes the processor priority (Figure 1.13) while executing the statement specified as its argument, and then resets the priority to its previous value. This prevents network device interrupts from occurring while the statement `mbstat.m_mtotypes[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.

92-97 The first function called by `m_retry` is `m_reclaim`. We'll see in Section 7.4 that each protocol can define a "drain" function to be called by `m_reclaim` when the system gets low on available memory. We'll also see in Figure 10.32 that when IP's drain function is called, all IP fragments waiting to be reassembled into IP datagrams are discarded. TCP's drain function does nothing and UDP doesn't even define a drain function.

98-102 Since there's a chance that more memory *might* be available after the call to `m_reclaim`, the `MGET` macro is called again, to try to obtain the mbuf. Before expanding the `MGET` macro (Figure 2.12), `m_retry` is defined to be a null pointer. This prevents an infinite loop if the memory still isn't available: the expansion of `MGET` will set `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.

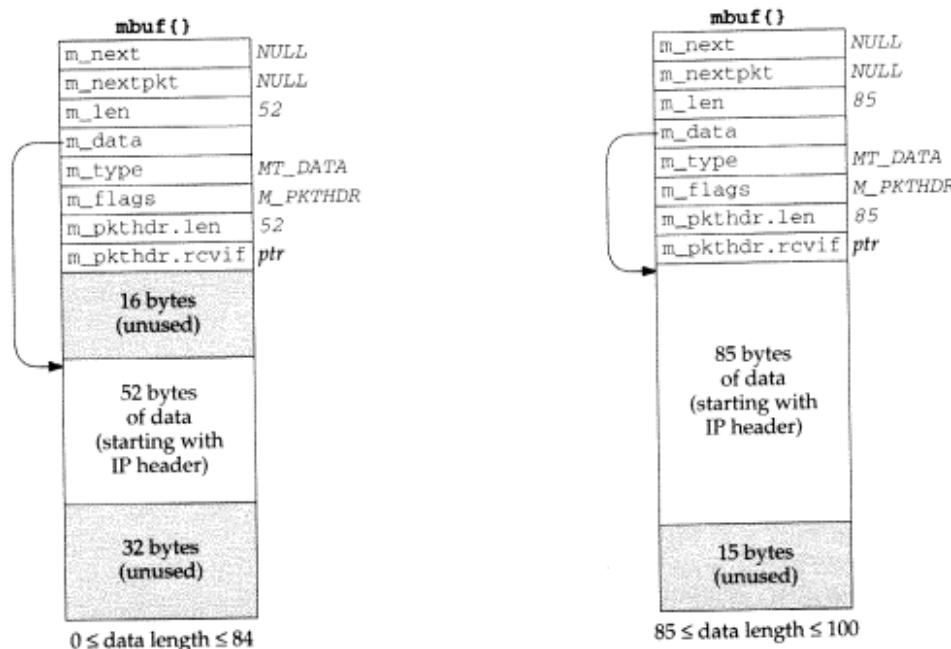


Figure 2.14 First two types of mbufs created by `m_devget`.

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.

- 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.
- Figure 2.15 shows the third type of mbuf created by `m_devget`. Two mbus 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.

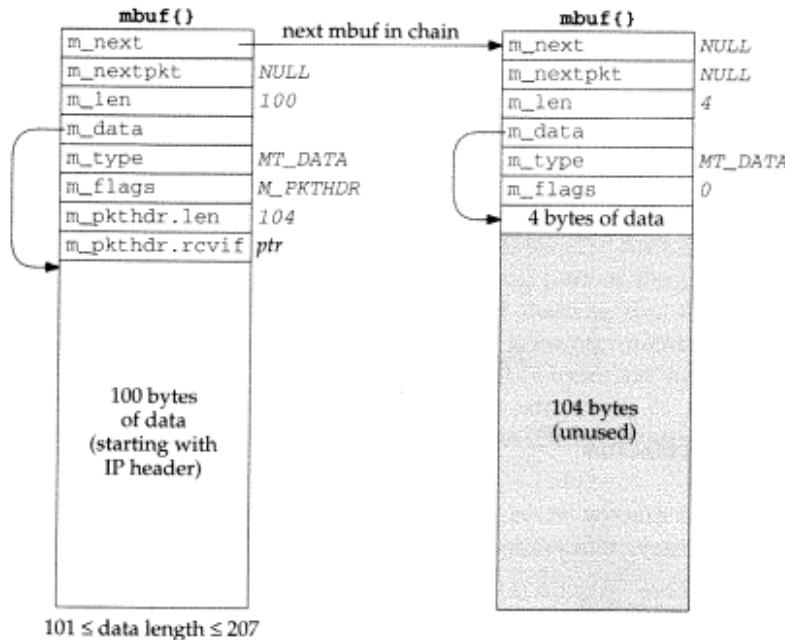


Figure 2.15 Third type of mbuf created by `m_devget`.

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.

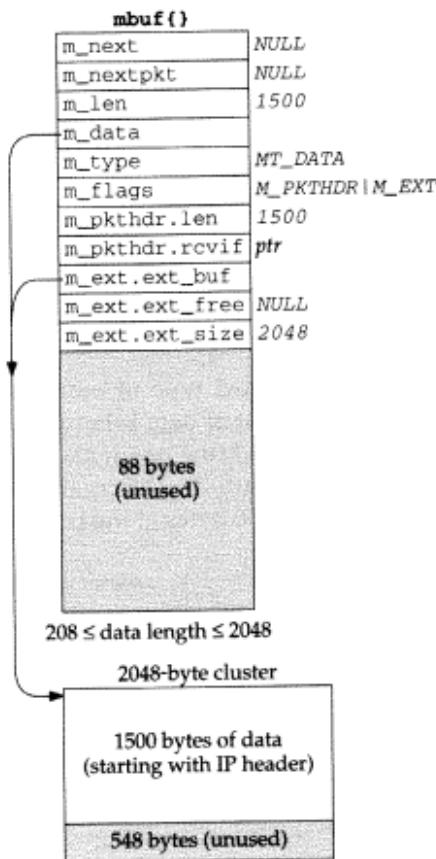


Figure 2.16 Fourth type of mbuf created by `m_devget`.

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

the amount of
ers are used.
-byte clus-
mbufs, each

```
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 (`MHLEN`). 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

complex

and casts

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 `MPFail` that is incremented each time `m_pullup` fails. On a Net/3 system that had received over 27 million IP datagrams, `MPFail` 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

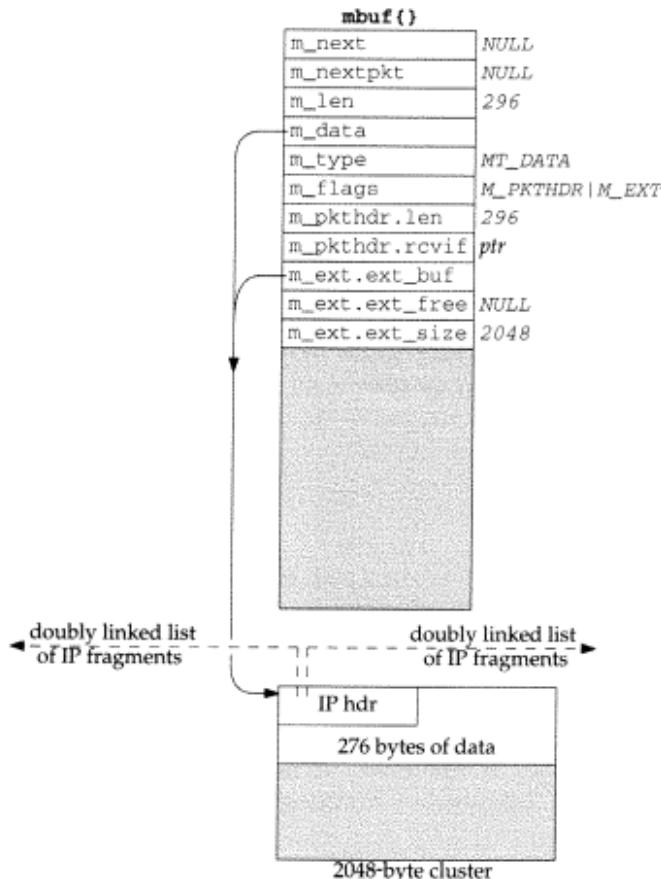


Figure 2.17 An IP fragment of length 296.

```

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

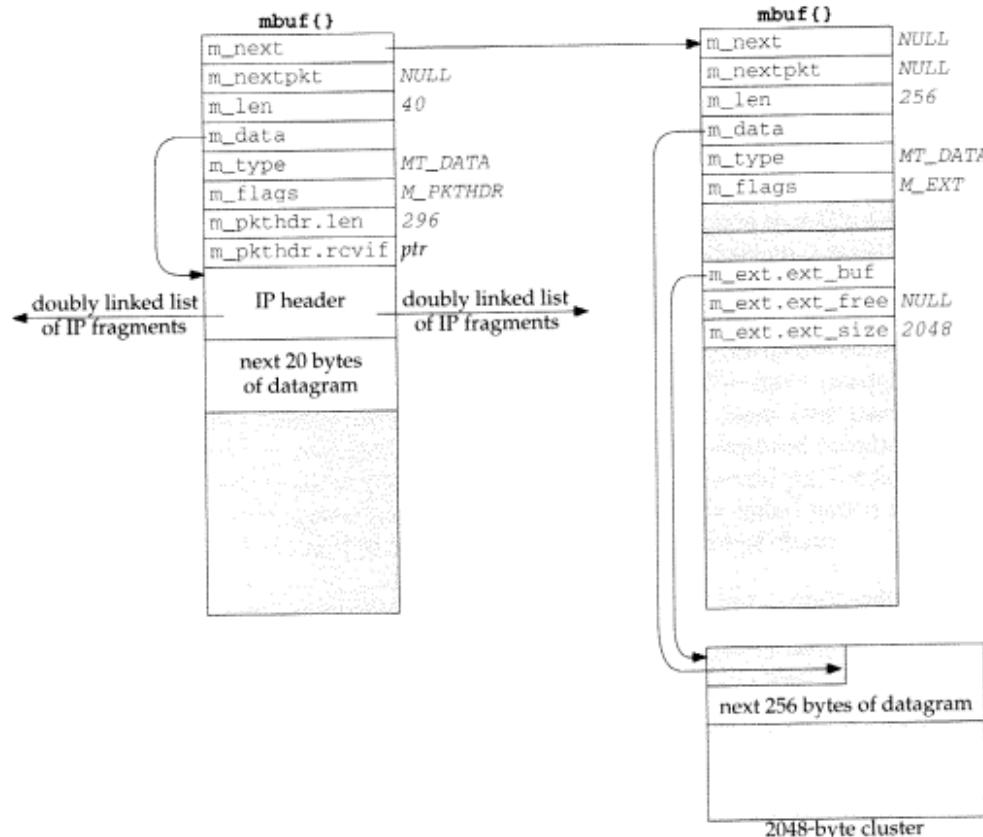


Figure 2.18 An IP fragment of length 296, after calling `m_pullup`.

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 <i>m->m_next</i> , 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.

MIT, and the
 using mbuf
 in the M_EXT
 the cluster's
 size count
 is pointed to
 MGET (Fig-
 ues to the
 sum for an
 also
 mom exists
 m_len) by
 pointer is set
 & mbuf is set
 Also, if a
 the packet
 an mbuf, into
 UDP headers
 are 1.8, caus-
 in the transi-
 ready existed
 ends for "core
 It was origi-
 system calls.
 the type of the

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

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.

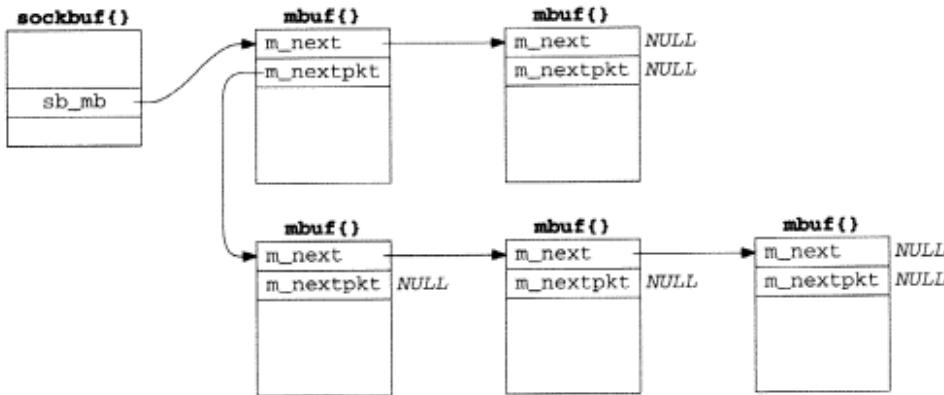


Figure 2.21 Linked list of mbuf chains with head pointer only.

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.

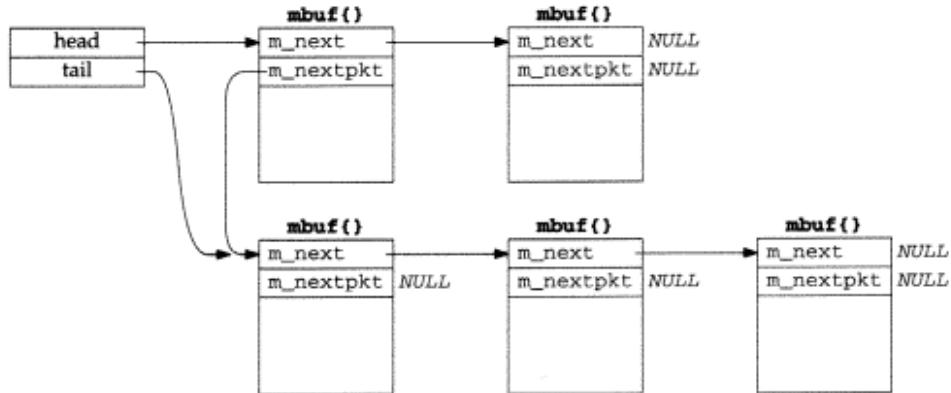


Figure 2.22 Linked list with head and tail pointers.

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).

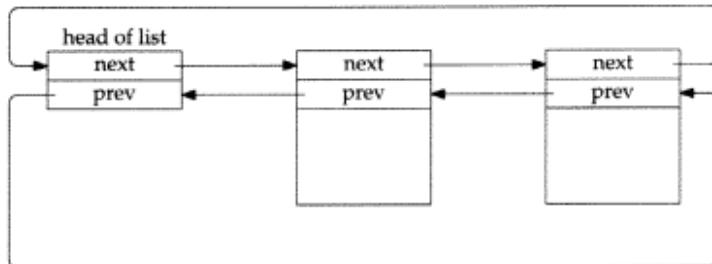


Figure 2.23 Doubly linked, circular list.

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`.

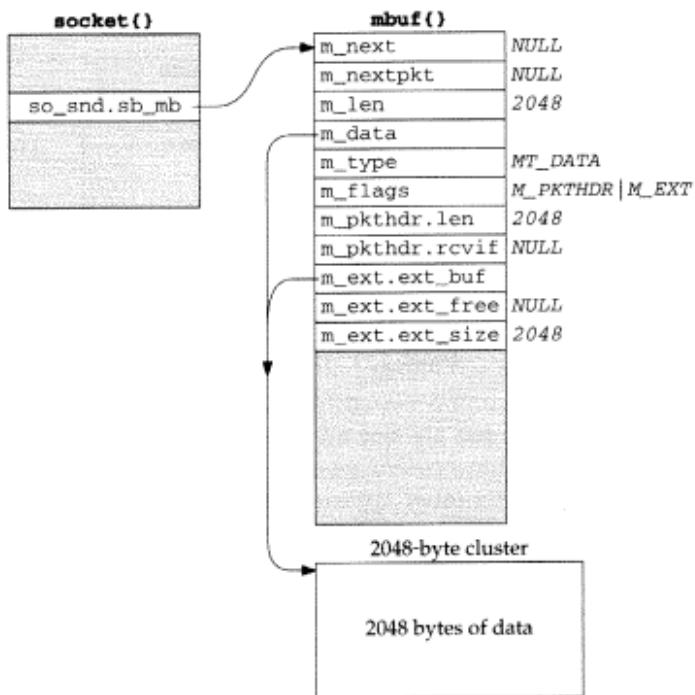


Figure 2.24 TCP socket send buffer containing 2048 bytes of data.

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.

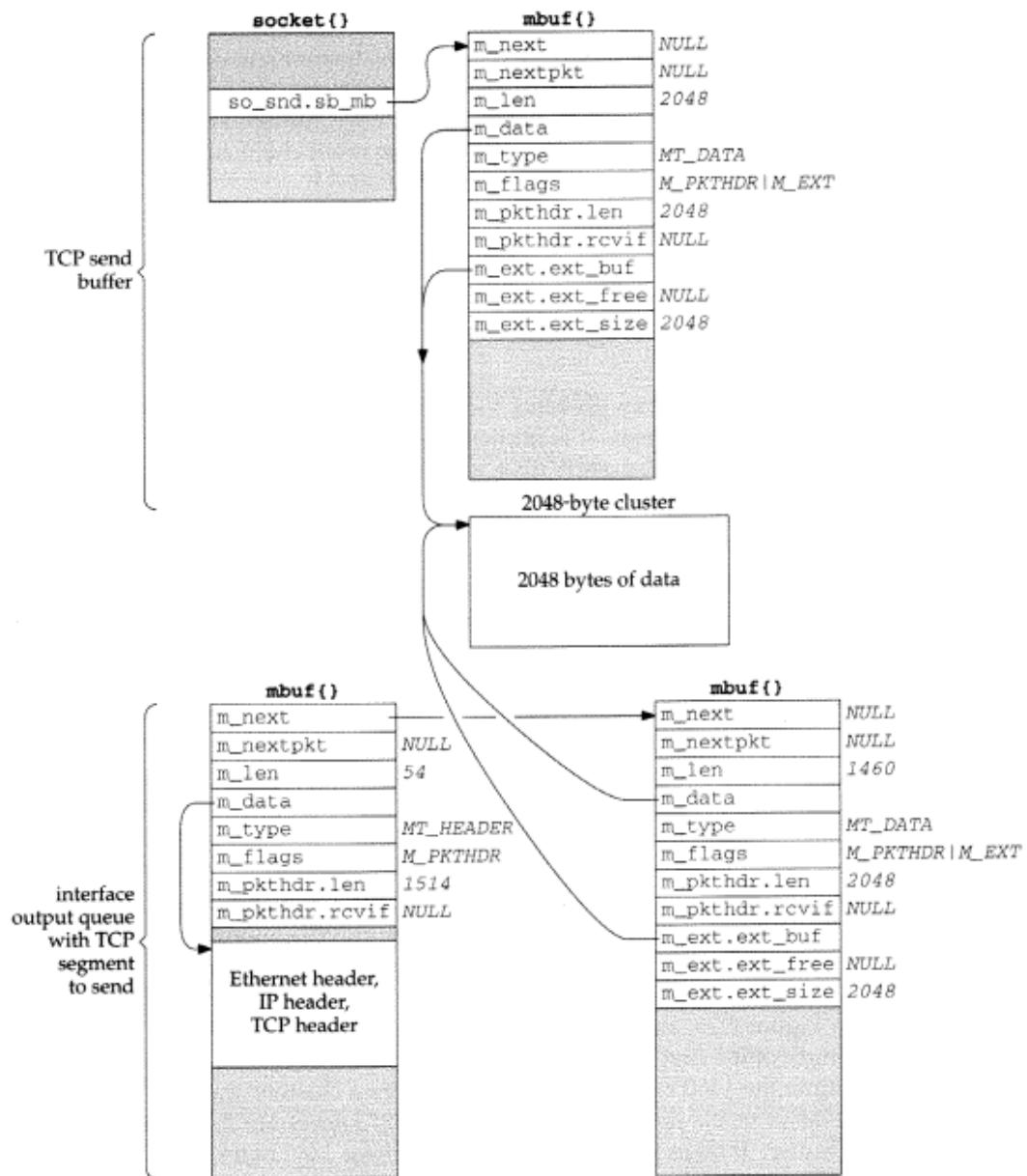


Figure 2.25 TCP socket send buffer and resulting segment on interface's output queue.

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.

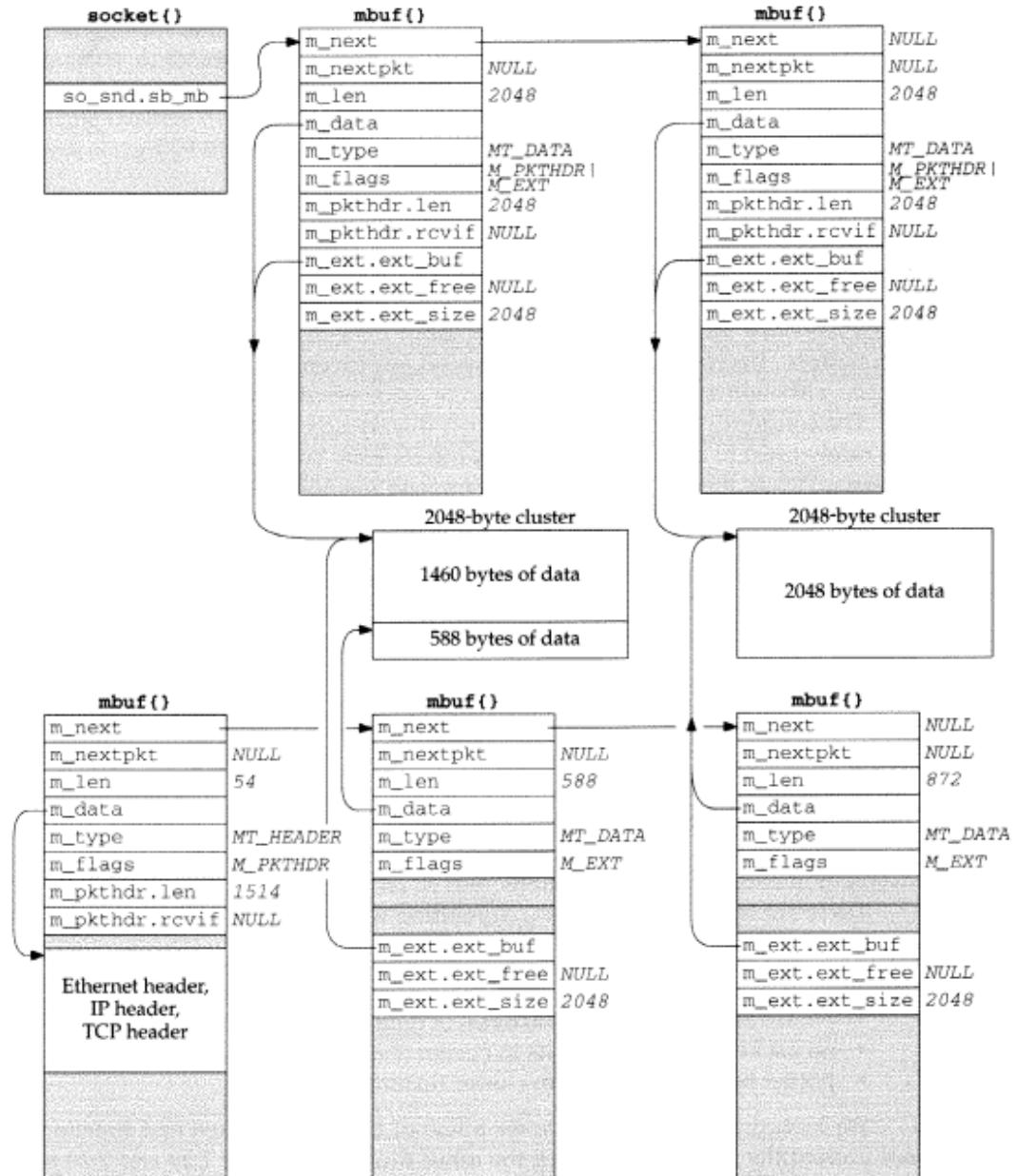


Figure 2.26 Mbuf chain to send next 1460-byte TCP segment.

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 second mbuf in the socket send buffer containing a packet header, even though it is not the first mbuf in the chain. This is a property of the `sosend` function, and this extraneous packet header is just ignored.

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

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

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.

packets as

headers and
is found in

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.

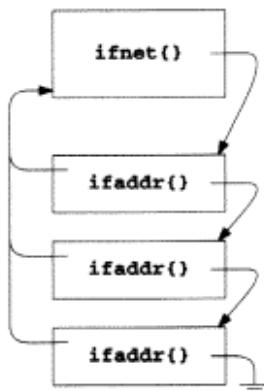


Figure 3.5 Each `ifnet` structure has a list of associated `ifaddr` structures.

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

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 *SIOCSIFFLAGS* commands described in Section 4.4.

<i>if_flags</i>	Kernel only	Description
<i>IFF_BROADCAST</i>	•	the interface is for a broadcast network
<i>IFF_MULTICAST</i>	•	the interface supports multicasting
<i>IFF_POINTOPOINT</i>	•	the interface is for a point-to-point network
<i>IFF_LOOPBACK</i>	•	the interface is for a loopback network
<i>IFF_OACTIVE</i>	•	a transmission is in progress
<i>IFF_RUNNING</i>	•	resources are allocated for this interface
<i>IFF_SIMPLEX</i>	•	the interface cannot receive its own transmissions
<i>IFF_LINK0</i>	see text	defined by device driver
<i>IFF_LINK1</i>	see text	defined by device driver
<i>IFF_LINK2</i>	see text	defined by device driver
<i>IFF_ALLMULTI</i>		the interface is receiving all multicast packets
<i>IFF_DEBUG</i>		debugging is enabled for the interface
<i>IFF_NOARP</i>		don't use ARP on this interface
<i>IFF_NOTRAILERS</i>		avoid using trailer encapsulation
<i>IFF_PROMISC</i>		the interface receives all network packets
<i>IFF_UP</i>		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_addrlen;       /* media address length */
94     u_char    ifi_hdrlen;        /* media header length */
95     u_long    ifi_mtu;          /* maximum transmission unit */
96     u_long    ifi_metric;        /* routing metric (external only) */
97     u_long    ifi_baudrate;      /* linespeed */

                                         /* other ifnet members */

138 #define if_mtu      if_data.ifi_mtu
139 #define if_type      if_data.ifi_type
140 #define if_addrlen   if_data.ifi_addrlen
141 #define if_hdrlen    if_data.ifi_hdrlen
142 #define if_metric    if_data.ifi_metric
143 #define if_baudrate  if_data.ifi_baudrate

```

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>IPT_OTHER</i>	unspecified
<i>IPT_ETHER</i>	Ethernet
<i>IPT_ISO88023</i>	IEEE 802.3 Ethernet (CMSA/CD)
<i>IPT_ISO88025</i>	IEEE 802.5 token ring
<i>IPT_FDDI</i>	Fiber Distributed Data Interface
<i>IPT_LOOP</i>	loopback interface
<i>IPT_SLIP</i>	serial line IP

Figure 3.9 if_type: data-link types.

93-94 *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).

95 *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.

96-97 *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

98-111 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 *ifsnd.ifq_drops*.

Change timestamp

112-113 *if_lastchange* records the last time any of the statistics were changed.

column.
depending
SLIP driver.

function
interface

Filter
received by
are passed

the hard-

if.h

ifnet
statistics

if.h

on lines 138

face. Fig-

```

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
                           interface */
109    u_long ifi_noproto; /* #packets destined for unsupported
                           protocol */
110    struct timeval ifi_lastchange; /* last updated */
111 } if_data;
112
113
114 #define if_ipackets if_data.ifi_ipackets
115 #define if_ierrors if_data.ifi_ierrors
116 #define if_opackets if_data.ifi_opackets
117 #define if_oerrors if_data.ifi_oerrors
118 #define if_collisions if_data.ifi_collisions
119 #define if_ibytes if_data.ifi_ibytes
120 #define if_obytes if_data.ifi_obytes
121 #define if_imcasts if_data.ifi_imcasts
122 #define if_omcasts if_data.ifi_omcasts
123 #define if_iqdrops if_data.ifi_iqdrops
124 #define if_noproto if_data.ifi_noproto
125 #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.

if.h

```

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

```

if.h

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.

if.h

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 };

```

if.h

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

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 };

```

*if.h**if.h*

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

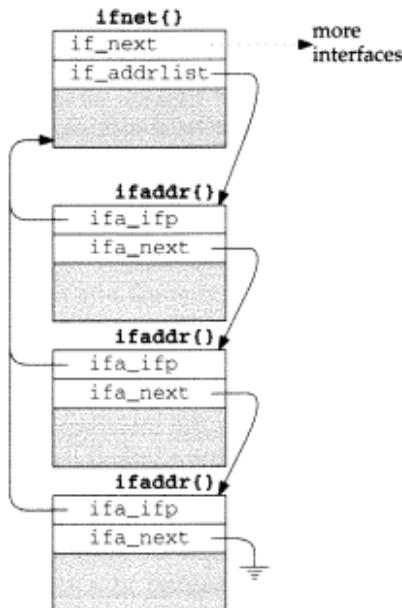


Figure 3.16 ifnet and ifaddr structures.

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 `SIOCDIFADDR` ioctl command. The `ifaddr` structures are reference-counted because they are shared by the interface and routing data structures.

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

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

Figure 3.17 sockaddr and osockaddr structures.

Figure 3.18 illustrates the organization of these structures.

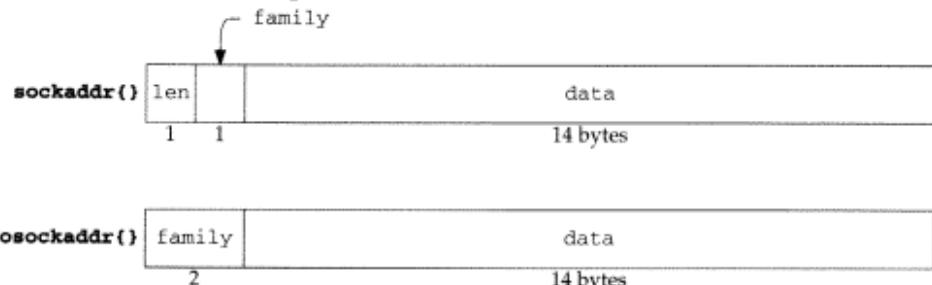


Figure 3.18 sockaddr and osockaddr structures (sa_ prefix dropped).

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.

<code>sa_family</code>	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

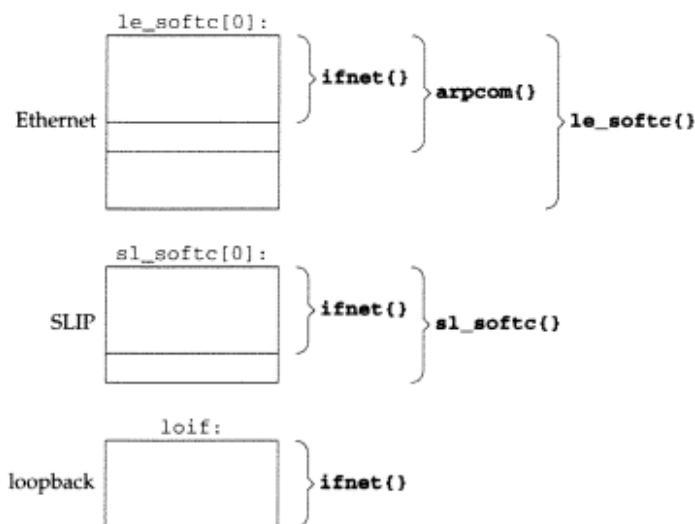


Figure 3.20 Arrangement of `ifnet` structures within device-dependent structures.

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.

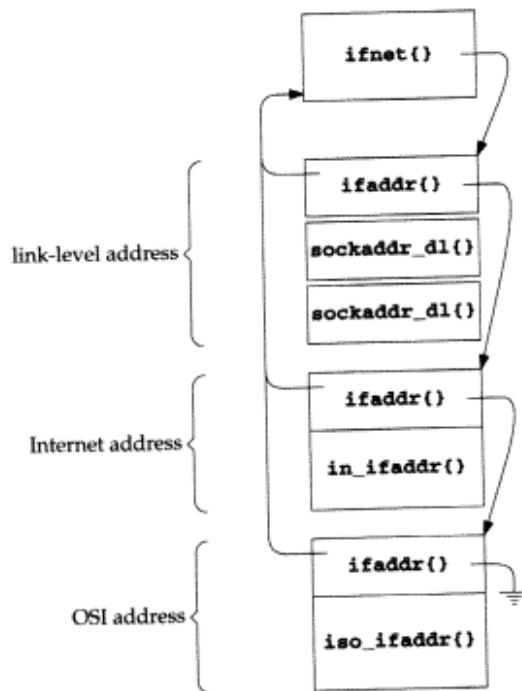


Figure 3.21 An interface address list containing link-level, Internet, and OSI addresses.

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.

init_main.c

```

70 main(framep)
71 void    *framep;
72 {

73     /* nonnetwork code */

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

74     /* 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);

75     /* nonnetwork code */

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

init_main.c

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).

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

    /* device-specific members */

95 } le_softc[NLE];
```

Figure 3.25 `le_softc` structure.

`le_softc` structure

An array of `le_softc` structures (with `NLE` elements) is declared in `if_le.c`.
 69-95 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.

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

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.

106-115 Figure 3.27 shows the initialization code for the LANCE Ethernet driver. The kernel calls leattach once for each LANCE card it finds in the system.

The single argument points to an hp_device structure, which contains HP-specific information since this driver is written for an HP workstation.

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

Copy the hardware address from the device

126-137 For the LANCE device, the Ethernet address assigned by the manufacturer is copied from the device to sc_addr (which is sc_ac.ac_enaddr—see Figure 3.26) one nibble (4 bits) at a time in this for loop.

testd 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).

```
if_le.c  
106 leattach(hd)  
107 struct hp_device *hd;  
108 {  
109     struct lereg0 *ler0;  
110     struct lereg2 *ler2;  
111     struct lereg2 *lmemem = 0;  
112     struct le_softc *le = &le_softc[hd->hp_unit];  
113     struct ifnet *ifp = &le->sc_if;  
114     char *cp;  
115     int i;  
  
116     /* device-specific code */  
  
126     /*  
127      * Read the ethernet address off the board, one nibble at a time.  
128      */  
129     cp = (char *) (lestd[3] + (int) hd->hp_addr);  
130     for (i = 0; i < sizeof(le->sc_addr); i++) {  
131         le->sc_addr[i] = (*++cp & 0xF) << 4;  
132         cp++;  
133         le->sc_addr[i] |= *++cp & 0xF;  
134         cp++;  
135     }  
136     printf("le%d: hardware address %s\n", hd->hp_unit,  
137           ether_sprintf(le->sc_addr));  
  
138     /* device-specific code */  
139  
150     ifp->if_unit = hd->hp_unit;  
151     ifp->if_name = "le";  
152     ifp->if_mtu = ETHERMTU;  
153     ifp->if_init = leinit;  
154     ifp->if_reset = lereset;  
155     ifp->if_ioctl = leioctl;  
156     ifp->if_output = ether_output;  
157     ifp->if_start = lestart;  
158     ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;  
159     bpfattach(&ifp->if_bpf, ifp, DLT_EN10MB, sizeof(struct ether_header));  
160     if_attach(ifp);  
161     return (1);  
162 }
```

if_le.c

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.

if_le.c

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_svar.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

within the
main calls
structure.

```

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

```

*if_sl.c***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).

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

if_sl.c

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_addrlen = 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.

41-56 The loopback `if_mtu` is set to 1536 bytes (LOMTU). In `if_flags`, `IFF_LOOPBACK` and `IFF_MULTICAST` are set. A loopback interface has no link header or hardware address, so `if_hdrlen` and `if_addrlen` are set to 0. `if_attach` finishes the initialization of the `ifnet` structure and `bpfattach` registers the loopback interface with BPF.

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

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`.

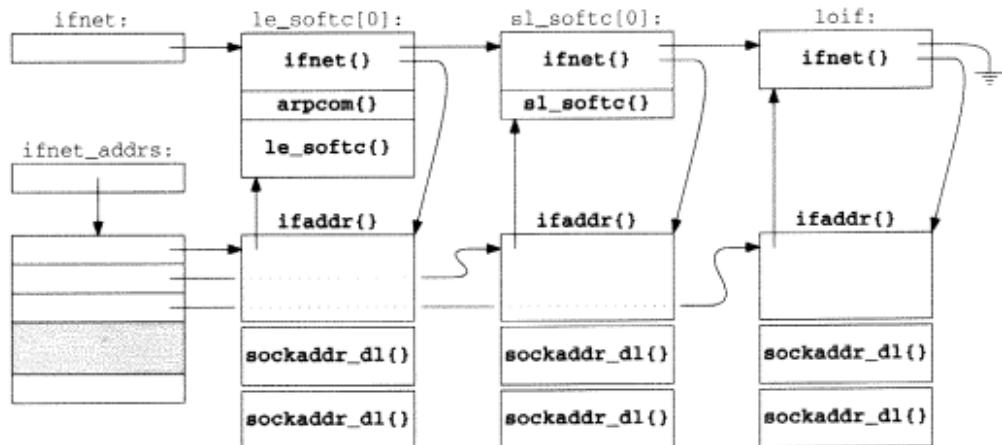


Figure 3.32 ifnet list.

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.

61–68 In addition to a numeric index, each interface has a text name formed from the `if_name` and `if_unit` members of the `ifnet` structure. For example, the first SLIP interface is called “`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.

59–74 `if_attach` has a single argument, `ifp`, a pointer to the `ifnet` structure that has been initialized by a network device driver. Net/3 keeps all the `ifnet` structures on a linked list headed by the global pointer `ifnet`. The while loop locates the end of the list and saves the address of the null pointer at the end of the list in `p`. After the loop, the new `ifnet` structure is attached to the end of the `ifnet` list, `if_index` is incremented, and the new index is assigned to `ifp->if_index`.

Resize `ifnet_addrs` array if necessary

75–85 The first time through `if_attach`, the `ifnet_addrs` array doesn't exist so space for 16 entries ($16 = 8 \ll 1$) 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_xrequest();

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 < \leq = 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

if.c

```

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

90     /* compute size of sockaddr_dl structure for this device */
91 #define _offsetof(t, m) ((int)((caddr_t)&((t *)0)->m))
92     masklen = _offsetof(struct sockaddr_dl, sdl_data[0]) +
93             unitlen + namelen;
94     socksize = masklen + ifp->if_addrflen;
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;

```

if.c

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_addrflen), 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.

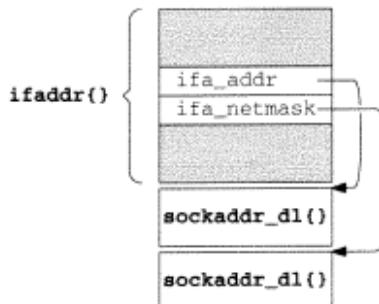


Figure 3.36 The link-level address and mask assigned during if_attach.

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)) {
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.

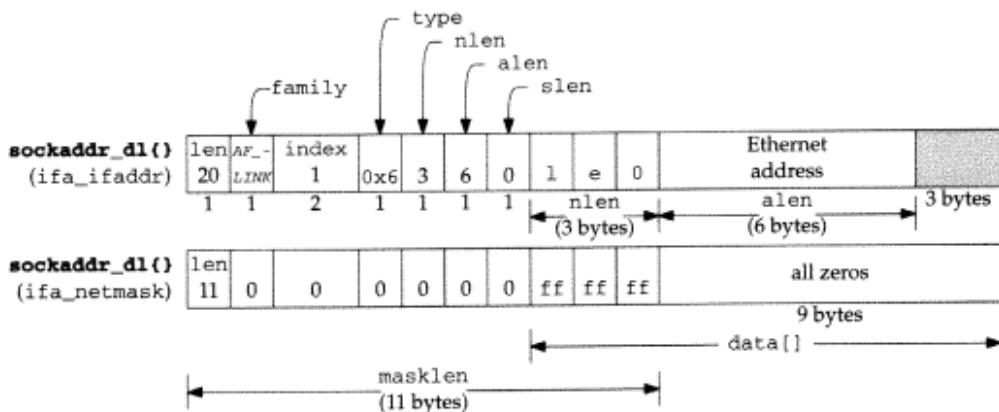


Figure 3.38 The initialized Ethernet `sockaddr_dl` structures (`sdl_` prefix omitted).

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`.

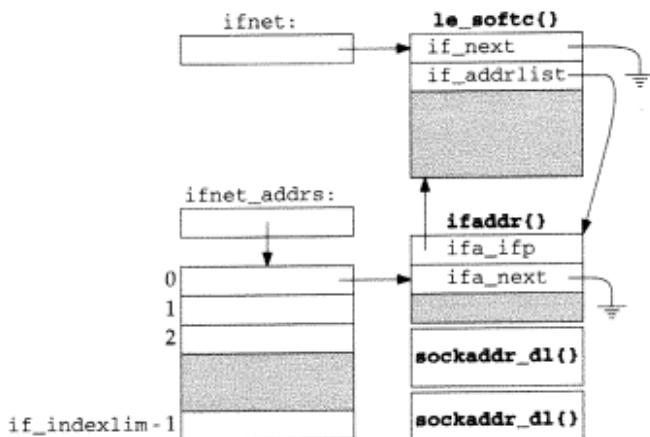


Figure 3.39 The `ifaddr` and `sockaddr_dl` structures after `if_attach` is called for the first time.

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     ifp->if_type = IFT_ETHER;
345     ifp->if_addrlen = 6;
346     ifp->if_hdrlen = 14;
347     ifp->if_mtu = ETHERMTU;
348     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
349         if ((sdl = (struct sockaddr_dl *) ifa->ifa_addr) &&
350             sdl->sdl_family == AF_LINK) {
351             sdl->sdl_type = IFT_ETHER;
352             sdl->sdl_alen = ifp->if_addrlen;
353             bcopy((caddr_t) ((struct arpcom *) ifp)->ac_enaddr,
354                   LLADDR(sdl), ifp->if_addrlen);
355             break;
356         }
357 }

```

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 arpcom 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.

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

if.c

if.c

Figure 3.42 ifinit function.

43-51 The for loop traverses the interface list and sets the maximum size of each interface output queue to 50 (ifqmaxlen) if it hasn't already been set by the interface's attach function.

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

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

```
if.c
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 }
```

if.c

if.c

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	leioctl	sliioctl	loioctl	handle ioctl commands from a process
if_reset	lereset			reset the device to a known state
if_watchdog				watch the device for failures or collect statistics

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 `leioctl`). `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 `ioctls` resides in two headers and three C files, which are listed in Figure 4.3.

File	Description
<code>netinet/if_ether.h</code> <code>net/if.h</code>	Ethernet structures ioctl command definitions
<code>net/if_ETHERSUBR.C</code> <code>hp300/dev/if_le.c</code> <code>net/if.c</code>	generic Ethernet functions LANCE Ethernet driver 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_igdrops	#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.

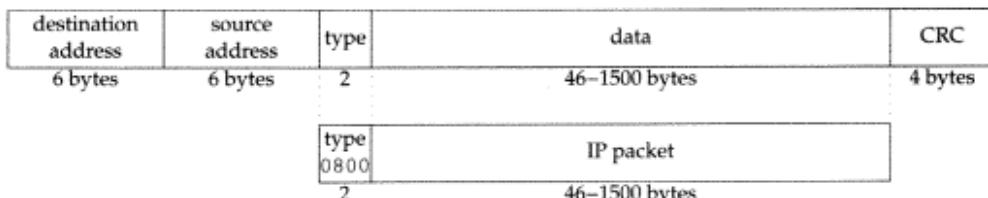


Figure 4.8 Ethernet encapsulation of an IP packet.

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.

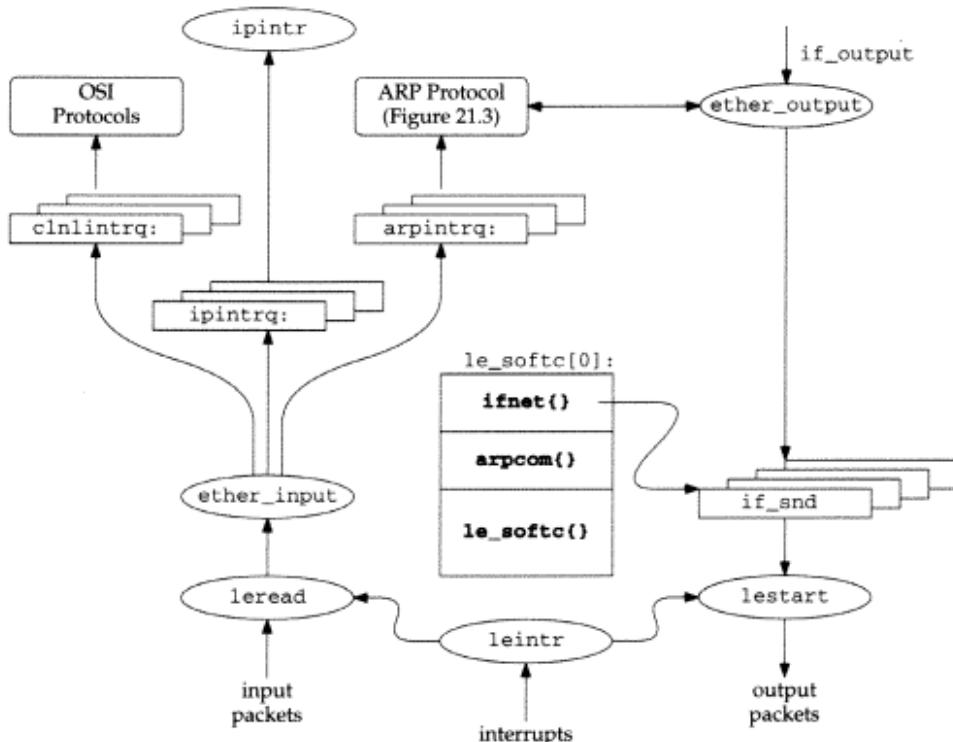


Figure 4.9 Ethernet device driver.

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 `lerestart`, 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.

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.

```

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.

```

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 (bcm((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     */
```

if_le.c

```

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.

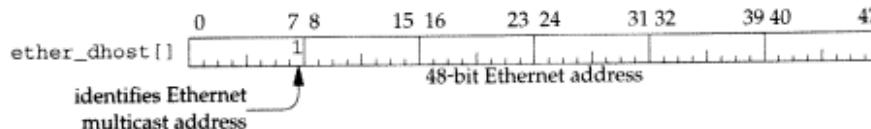


Figure 4.12 Testing for an Ethernet multicast address.

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 `leread` 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 `leread` 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, `leread` 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 (bcmpl(caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
213         sizeof(etherbroadcastaddr)) == 0)
214         m->m_flags |= M_BCAST;
215     else if (eh->ether_dhost[0] & 1)
216         m->m_flags |= M_MCAST;
217     if (m->m_flags & (M_BCAST | M_MCAST))
218         ifp->if_imcasts++;
219
220     switch (eh->ether_type) {
221     case ETHERTYPE_IP:
222         schednetisr(NETISR_IP);
223         inq = &ipintrq;
224         break;
225
226     case ETHERTYPE_ARP:
227         schednetisr(NETISR_ARP);
228         inq = &arpintrq;
229         break;
230
231     default:
232         if (eh->ether_type > ETHERMTU) {
233             m_freem(m);
234             return;
235         }
236
237     /* OSI code */
238
239     }
240
241     s = splimp();
242     if (IF_QFULL(inq)) {
243         IF_DROP(inq);
244         m_freem(m);
245     } else
246         IF_ENQUEUE(inq, m);
247     splx(s);
248 }

```

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 queuing 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 `schednetisr` 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 rtentry *rt0;
55 {
56     short type;
57     int s, error = 0;
58     u_char edst[6];
59     struct mbuf *m = m0;
60     struct rtentry *rt;
61     struct mbuf *mcopy = (struct mbuf *) 0;
62     struct ether_header *eh;
63     int off, len = m->m_pkthdr.len;
64     struct arpcom *ac = (struct arpcom *) ifp;

65     if ((ifp->if_flags & (IFF_UP | IFF_RUNNING)) != (IFF_UP | IFF_RUNNING))
66         senderr(ENETDOWN);
67     ifp->if_lastchange = time;
68     if (rt = rt0) {
69         if ((rt->rt_flags & RTF_UP) == 0) {
70             if (rt0 = rt = rtalloc1(dst, 1))
71                 rt->rt_refcnt--;
72             else
73                 senderr(EHOSTUNREACH);
74         }
75         if (rt->rt_flags & RTF_GATEWAY) {
76             if (rt->rt_gwroute == 0)
77                 goto lookup;
78             if (((rt = rt->rt_gwroute)->rt_flags & RTF_UP) == 0) {
79                 rtfree(rt);
80                 rt = rt0;
81             lookup: rt->rt_gwroute = rtalloc1(rt->rt_gateway, 1);
82                 if ((rt = rt->rt_gwroute) == 0)
83                     senderr(EHOSTUNREACH);
84             }
85         }
86         if (rt->rt_flags & RTF_REJECT)
87             if (rt->rt_rmx.rmx_expire == 0 ||
88                 time.tv_sec < rt->rt_rmx.rmx_expire)
89                 senderr(rt == rt0 ? EHOSTDOWN : EHOSTUNREACH);
90     }

```

if_ETHERSUBR.C

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            /* ISO code */
103
104        case AF_UNSPEC:
105            eh = (struct ether_header *) dst->sa_data;
106            bcopy((caddr_t) eh->ether_dhost, (caddr_t) edst, sizeof(edst));
107            type = eh->ether_type;
108            break;
109
110        default:
111            printf("%s%d: can't handle af%d\n", ifp->if_name, ifp->if_unit,
112                  dst->sa_family);
113            senderr(EAFNOSUPPORT);
114    }

```

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));

```

if_ETHERSUBR.C

if_ETHERSUBR.C

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.

it checks if it receive its packet. After we. This is copy of the

arrived on the

and type points to an assigns the ET_H) because

$\text{ether}_\text{output}$

net frame is

$\text{if}_\text{ethersubr.c}$

$\text{if}_\text{ethersubr.c}$

cessed as if it
ck interface

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 $\text{igmp}_\text{sendreport}$ in Section 13.6).

$\text{ether}_\text{output}$ 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. $\text{ether}_\text{output}$ overwrites the source address the caller may have specified in the $\text{ether}_\text{header}$ 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();                                     if_etheroutput.c
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->ifobytes += 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_free(m);
189     return (error);
190 }
```

if_etheroutput.c

Figure 4.18 $\text{ether}_\text{output}$ function: output queueing.

168–185 If the output queue is full, $\text{ether}_\text{output}$ 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.

lrestart Function

The lrestart 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 lrestart 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 lrestart to dequeue and transmit the next frame. Once started, the protocol layer can queue frames without calling lrestart since the driver dequeues and transmits frames until the queue is empty.

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

Interface must be initialized

325-333 If the interface is not initialized, lrestart 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, lrestart 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 lrestart 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, lrestart 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     do {                                         /* device-specific code */
337
338         IF_DEQUEUE(&le->sc_if.if_snd, m);          /* device-specific code */
339         if (m == 0)
340             return (0);
341         len = leput(le->sc_r2->ler2_tbuf[le->sc_tmd], m);
342         /*
343          * If bpf is listening on this interface, let it
344          * see the packet before we commit it to the wire.
345          */
346         if (ifp->if_bpf)
347             bpf_tap(ifp->if_bpf, le->sc_r2->ler2_tbuf[le->sc_tmd],
348                     len);
349
350
351         } while (++le->sc_txcnt < LETBUF);
352         le->sc_if.if_flags |= IFF_OACTIVE;
353     return (0);
354 }

```

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>ifiocctl</code>	get interface flags
<code>SIOCGIFMETRIC</code>	<code>struct ifreq *</code>	<code>ifiocctl</code>	get interface metric
<code>SIOCSIFFLAGS</code>	<code>struct ifreq *</code>	<code>ifiocctl</code>	set interface flags
<code>SIOCSIFMETRIC</code>	<code>struct ifreq *</code>	<code>ifiocctl</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.

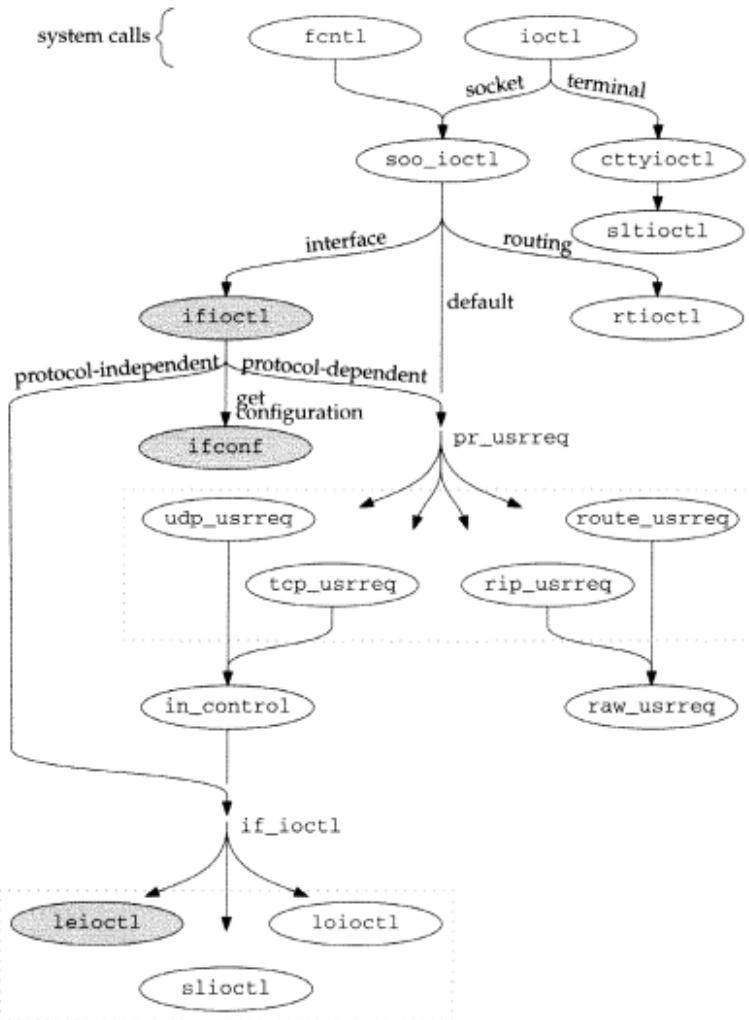


Figure 4.21 ioctl functions described in this chapter.

ifioctl Function

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

```
if.c
394 int
395 ifioctl(so, cmd, data, p)
396 struct socket *so;
397 int cmd;
398 caddr_t data;
399 struct proc *p;
400 {
401     struct ifnet *ifp;
402     struct ifreq *ifr;
403     int error;
404     if (cmd == SIOCGIFCONF)
405         return (ifconf(cmd, data));
406     ifr = (struct ifreq *) data;
407     ifp = ifunit(ifr->ifr_name);
408     if (ifp == 0)
409         return (ENXIO);
410     switch (cmd) {
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 }
```

if.c

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., "s10", "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 };

```

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 };

```

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.

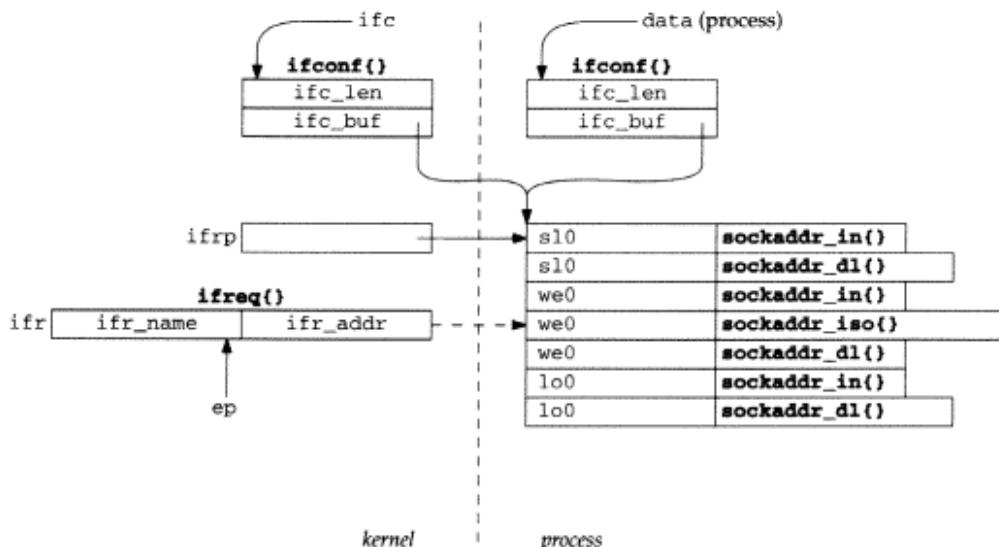


Figure 4.25 ifconf data structures.

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. ifp 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. 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                                if.c
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;

```

ulated by

```

473     ifrp = ifc->ifc_req;
474     ep = ifr.ifr_name + sizeof(ifr.ifr_name) - 2;
475     for (; space > sizeof(ifr) && 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 `ifrp` 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.

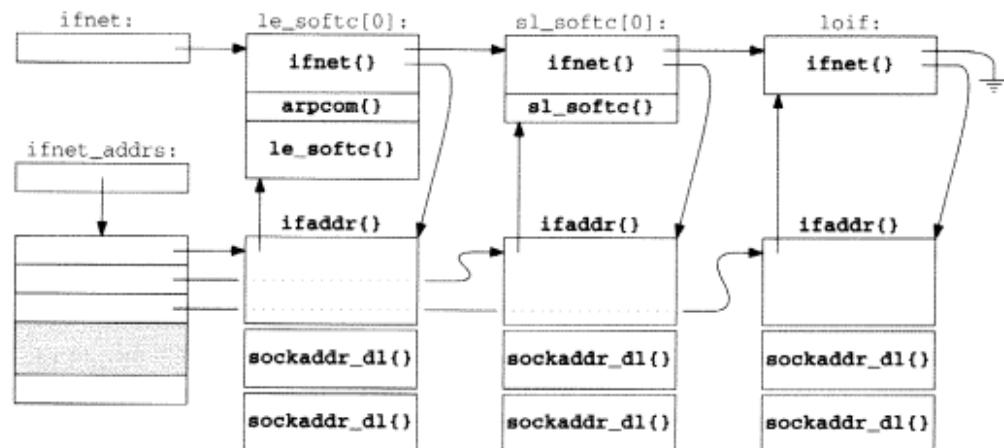


Figure 4.27 Interface and address data structures.

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 (3×36) 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.

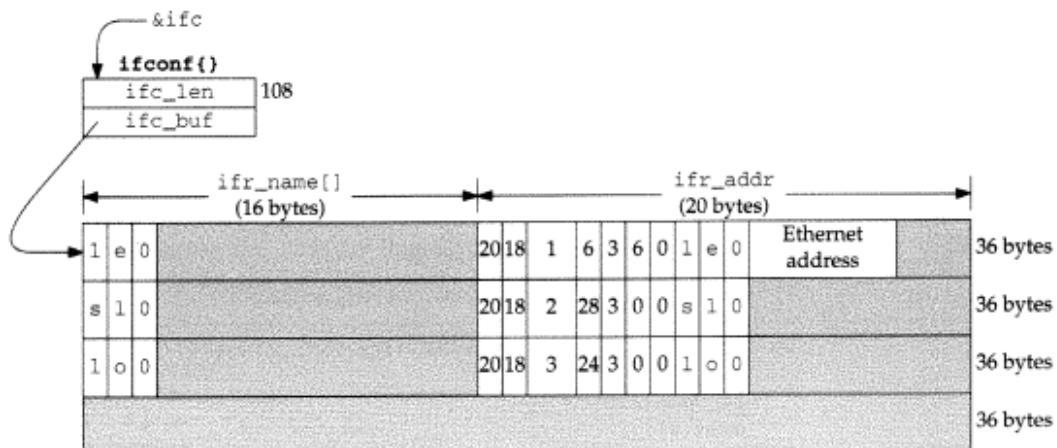


Figure 4.28 Data returned by the SIOCGIFCONF command.

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_slen` (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_slen` 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 `ifiioctl` function. Figure 4.29 shows the case statements for these commands.

`SIOCGIFFLAGS` and `SIOCGIFMETRIC`

For the two `SIOCGxxx` commands, `ifiioctl` 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).

```

if.c
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., `leiioctl` for the LANCE driver—Figure 4.31).

if.c

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

if.c

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 slioctl and loioctl functions return EINVAL for this command, which is ignored by ifioctl. Figure 4.31 shows the leioctl function and SIOCSIFFLAGS processing of the LANCE Ethernet driver.

```
if_le.c
614 leioctl(ifp, cmd, data)
615 struct ifnet *ifp;
616 int cmd;
617 caddr_t data;
618 {
619     struct ifaddr *ifa = (struct ifaddr *) data;
620     struct le_softc *le = &le_softc[ifp->if_unit];
621     struct lereg1 *ler1 = le->sc_r1;
622     int s = splimp(), error = 0;
623
624     switch (cmd) {
625
626         /* SIOCSIFADDR code (Figure 6.28) */
627
628     case SIOCSIFFLAGS:
629         if (((ifp->if_flags & IFF_UP) == 0 &&
630             ifp->if_flags & IFF_RUNNING) ||
631             LERDWR(le->sc_r0, LE_STOP, ler1->ler1_rdp));
632             ifp->if_flags |= ~IFF_RUNNING;
633         } else if ((ifp->if_flags & IFF_UP &&
634             (ifp->if_flags & IFF_RUNNING) == 0)
635             leinit(ifp->if_unit);
636         /*
637          * If the state of the promiscuous bit changes, the interface
638          * must be reset to effect the change.
639          */
640         if (((ifp->if_flags ^ le->sc_iflags) & IFF_PROMISC) &&
641             (ifp->if_flags & IFF_RUNNING)) {
642             le->sc_iflags = ifp->if_flags;
643             lereset(ifp->if_unit);
644             lestart(ifp);
645         }
646         break;
647
648         /* SIOCADDMULTI and SIOCDELMULTI code (Figure 12.31) */
649
650     default:
651         error = EINVAL;
652     }
653     splx(s);
654     return (error);
655 }
```

if_le.c

Figure 4.31 leioctl function: SIOCSIFFLAGS.

calls the
ioctl and
ifioctl.
the LANCE

— if_le.c

614–623 `leioctl` 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 `ifioctl` calls `leioctl`, 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 `ioctl` commands that access the interface-layer data structures.

Exercises

- 4.1 In `leread`, 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 `leread` 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
if_init	leinit			initialize hardware
if_output	ether_output	sloutput	looutput	accept and queue packet for transmission
if_start	lrestart			begin transmission of frame
if_done				output complete (unused)
if_ioctl	leioctl	sliioctl	loioctl	handle ioctl commands from a process
if_reset	lreset			reset the device to a known state
if_watchdog				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
<code>net/if_slvar.h</code>	SLIP definitions
<code>net/if_sl.c</code>	SLIP driver functions
<code>net/if_loop.c</code>	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
<code>sl_softc</code>	<code>struct sl_softc []</code>	SLIP interface
<code>loif</code>	<code>struct ifnet</code>	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
<code>tk_nin</code>	#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.

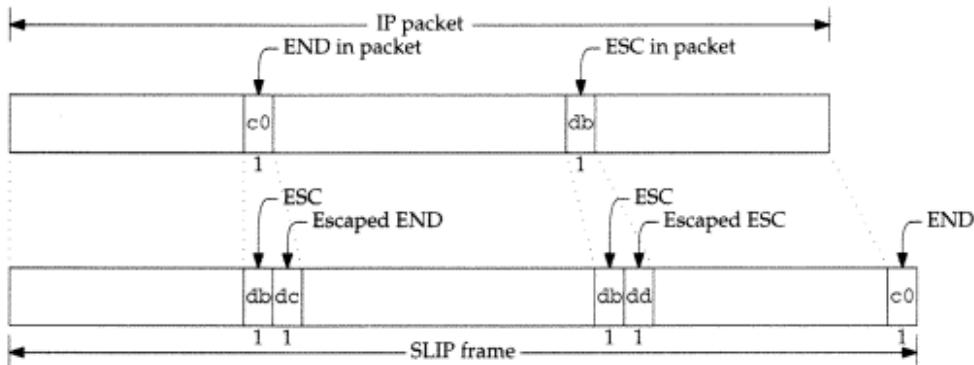


Figure 5.5 SLIP encapsulation of an IP packet.

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.

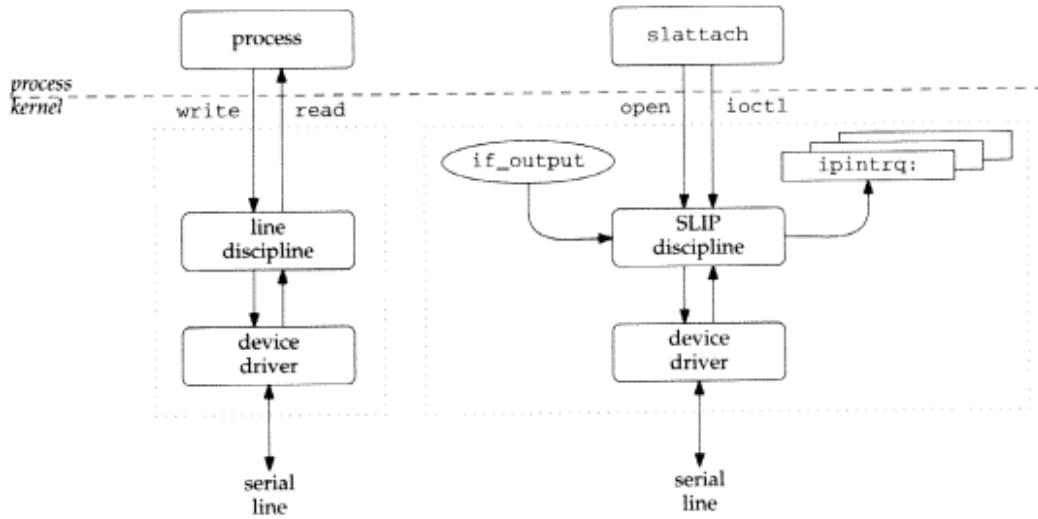


Figure 5.6 The SLIP interface as a line discipline.

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 <code>mbuf</code> 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
sltioclt		•	process TTY <code>ioctl</code> commands
slstart	•	•	dequeue packet and begin transmitting data on TTY device
slingput	•	•	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.

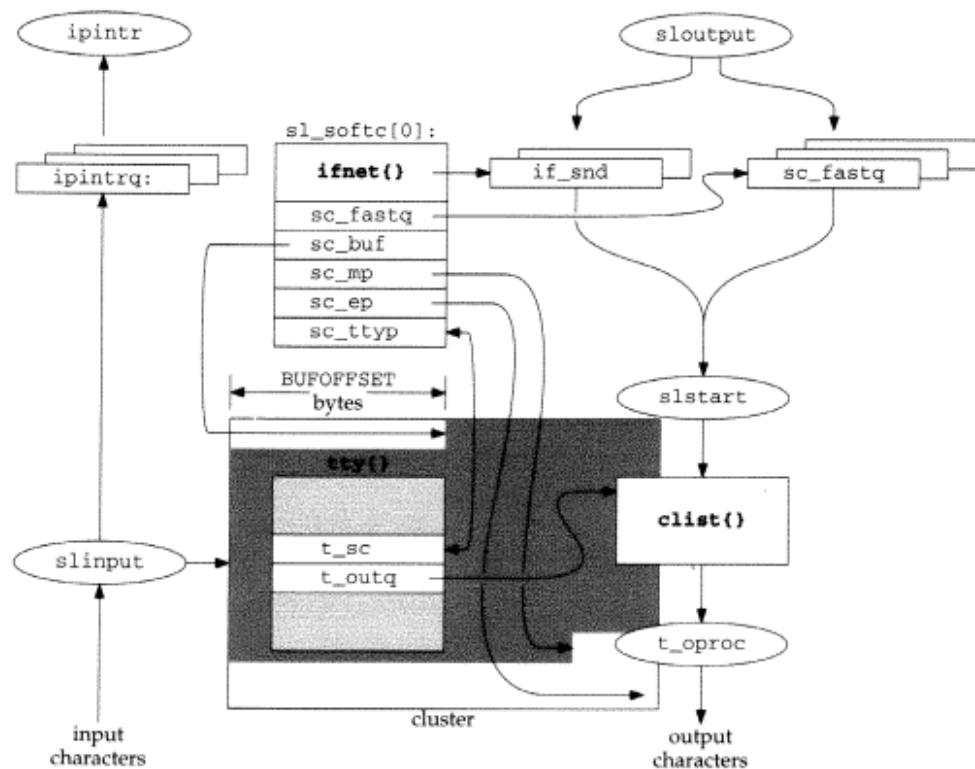


Figure 5.8 SLIP device driver.

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 illustrated the two links between the structures.

Figure 5.8 contains a lot of information:

- The network interface is represented by the `sl_softc` structure and the TTY device by the `tty` structure.
- Incoming bytes are stored in the cluster (shown behind the `tty` structure). When a complete SLIP frame is received, the enclosed IP packet is put on the `ipintrq` by `slinput`.
- Outgoing packets are dequeued from `if_snd` or `sc_fastq`, converted to SLIP frames, and passed to the TTY device by `slstart`. The TTY buffers outgoing bytes in the `cist` structure. The `t_oproc` function drains and transmits the bytes held in the `cist` structure.

SLIP Initialization: `slopen` and `slinit`

We discussed in Section 3.7 how `slattach` initializes the `sl_softc` structures. The interface remains initialized but inoperative until a program (usually `slattach`) opens a TTY device (e.g., `/dev/tty01`) and issues an `ioctl` command to replace the standard line discipline with the SLIP discipline. At this point the TTY subsystem calls the line discipline's open function (in this case `slopen`), which establishes the association between a particular TTY device and a particular SLIP interface. `slopen` is shown in Figure 5.9.

```

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
191     if (error = suser(p->p_ucred, &p->p_acflag))
192         return (error);
193
194     if (tp->t_line == SLIPDISC)
195         return (0);
196
197     for (nsl = NSL, sc = sl_softc; --nsl >= 0; sc++)
198         if (sc->sc_ttyp == NULL) {
199             if (slinit(sc) == 0)
200                 return (ENOBUFS);
201             tp->t_sc = (caddr_t) sc;
202             sc->sc_ttyp = tp;
203             sc->sc_if.if_baudrate = tp->t_ospeed;
204             ttyflush(tp, FREAD | FWRITE);
205         }
206     return (ENXIO);
207 }
```

if_sl.c

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 `slinit` (Figure 5.10), joins the `tty` and `sl_softc` structures by `t_sc` and `sc_ttyp`, and copies the TTY output speed (`t_ospeed`) into the SLIP interface. `ttyflush` discards any pending input or output data in the TTY queues. `slopen` returns `ENXIO` if a SLIP interface structure is not available, or 0 if it was successful.

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
162     if (sc->sc_ep == (u_char *) 0) {
163         MCLALLOC(p, M_WAIT);
164         if (p)
165             sc->sc_ep = (u_char *) p + SLBUFSIZE;
166         else {
167             printf("sl%d: can't allocate buffer\n", sc - sl_softc);
168             sc->if_if_flags &= ~IFF_UP;
169             return (0);
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 }
```

if_sl.c

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
<i>BUFOFFSET + SLMAX = SLBUFSIZE = MCLBYTES</i>		

Figure 5.11 SLIP constants.

SLIP Input Processing: *sliinput*

The TTY device driver delivers incoming characters to the SLIP line discipline one at a time by calling *sliinput*. Figure 5.12 shows the *sliinput* function but omits the end-of-frame processing, which is discussed separately.

```

527 void
528 sliinput(c, tp)
529 int    c;
530 struct tty *tp;
531 {
532     struct sl_softc *sc;
533     struct mbuf *m;
534     int    len;
535     int    s;
536     u_char chdr[CHDR_LEN];

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

```

```

553     case TRANS_FRAME_END:
554         if (sc->sc_escape)
555             c = FRAME_END;
556         break;
557     case FRAME_ESCAPE:
558         sc->sc_escape = 1;
559         return;
560     case FRAME_END:
561
562         /* FRAME_END code (Figure 5.13) */
563
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 *sliinput* 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 *sliinput* returns.

If the cluster is full, the character is discarded and *sliinput* 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 *sliinput* returns.

Figure 5.13 shows the FRAME_END code omitted from Figure 5.12.

if_sl.c

```

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_UNCOMPRESSED_TCP && len >= 40) {
599             len = sl_uncompress_tcp(&sc->sc_buf, len,
600                                     (u_int) c, &sc->sc_comp);
601             if (len <= 0)
602                 goto error;
603             sc->sc_if.if_flags |= SC_COMPRESS;
604         } else
605             goto error;
606     }
607     if (sc->sc_bpf) {
608         /*
609          * Put the SLIP pseudo-link header" in place.
610          * We couldn't do this any earlier since
611          * decompression probably moved the buffer
612          * pointer. Then, invoke BPF.
613          */
614     u_char *hp = sc->sc_buf - SLIP_HDRLEN;
615
616     hp[SLX_DIR] = SLIPDIR_IN;
617     bcopy(chdr, &hp[SLX_CHDR], CHDR_LEN);
618     bpf_tap(sc->sc_bpf, hp, len + SLIP_HDRLEN);
619     }
620     m = sl_btom(sc, len);
621     if (m == NULL)
622         goto error;
623
624     sc->sc_if.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, `sliinput` 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, `sl1_uncompress_tcp` attempts to uncompress the packet. If compression is not enabled, auto-enable compression is on, and if the packet is large enough `sl1_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 `sliinput`.

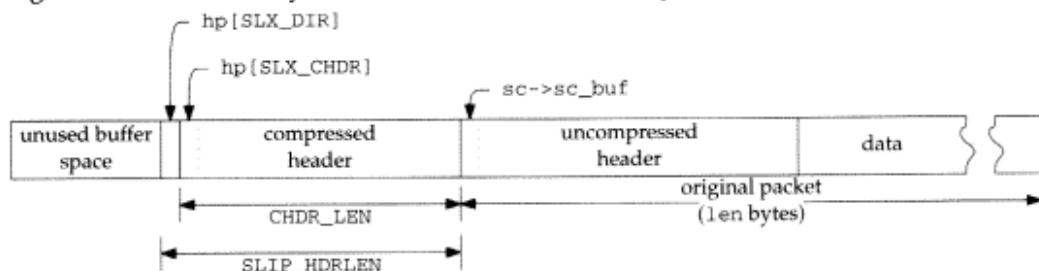


Figure 5.14 SLIP packet in BPF format.

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 `ipintraq`, 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).

```

259 int
260 sloutput(ifp, m, dst, rtp)
261 struct ifnet *ifp;
262 struct mbuf *m;
263 struct sockaddr *dst;
264 struct rtentry *rtp;
265 {
266     struct sl_softc *sc = &sl_softc[ifp->if_unit];
267     struct ip *ip;
268     struct ifqueue *ifq;
269     int     s;
270
271     /*
272      * Cannot happen (see slioctl). Someday we will extend
273      * the line protocol to support other address families.
274      */
275     if (dst->sa_family != AF_INET) {
276         printf("sl%d: af%d not supported\n", sc->sc_if.if_unit,
277                dst->sa_family);
278         m_freem(m);
279         sc->sc_if.if_noproto++;
280         return (EAFNOSUPPORT);
281     }
282     if (sc->sc_ttyp == NULL) {
283         m_freem(m);
284         return (ENETDOWN); /* sort of */
285     }
286     if ((sc->sc_ttyp->t_state & TS_CARR_ON) == 0 &&
287         (sc->sc_ttyp->t_cflag & CLOCAL) == 0) {
288         m_freem(m);
289         return (EHOSTUNREACH);
290     }
291     ifq = &sc->sc_if.if_snd;
292     ip = mtod(m, struct ip *);
293     if (sc->sc_if.if_flags & SC_NOICMP && ip->ip_p == IPPROTO_ICMP) {
294         m_freem(m);
295         return (ENETRESET); /* XXX ? */
296     }
297     if (ip->ip_tos & IPTOS_LOWDELAY)
298         ifq = &sc->sc_fastq;
299     s = splimp();
300     if (IF_QFULL(ifq)) {
301         IF_DROP(ifq);
302         m_freem(m);
303         splx(s);
304         sc->sc_if.if_oerrors++;
305         return (ENOBUFS);
306     }

```

```

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) sloutput calls slstart to initiate transmission of the packet.

SLIP increments if_oerrors if the interface queue is full; ether_output does not.

Unlike the Ethernet output function (ether_output), sloutput does not construct a data-link header for the outgoing packet. Since the only other system on a SLIP network is at the other end of the serial link, there is no need for hardware addresses or a protocol, such as ARP, to convert between IP addresses and hardware addresses. Protocol identifiers (such as the Ethernet type field) are also superfluous, since a SLIP link carries only IP packets.

slstart Function

In addition to the call by sloutput, the TTY device driver calls slstart when it drains its output queue and needs more bytes to transmit. The TTY subsystem manages its queues through a *clist* structure. In Figure 5.8 the output *clist* *t_outq* is shown below slstart and above the device's *t_oproc* function. slstart adds bytes to the queue, while *t_oproc* drains the queue and transmits the bytes.

The slstart function is shown in Figure 5.16.

318-358 When slstart is called, *tp* points to the device's *tty* structure. The body of slstart consists of a single for loop. If the output queue *t_outq* is not empty, slstart calls the output function for the device, *t_oproc*, which transmits as many bytes as the device will accept. If more than 100 bytes (SLIP_HIWAT) remain in the TTY output queue, slstart returns instead of adding another packet's worth of bytes to the queue. The output device generates an interrupt when it has transmitted all the bytes, and the TTY subsystem calls slstart when the output list is empty.

If the TTY output queue is empty, a packet is dequeued from *sc_fastq* or, if *sc_fastq* is empty, from the *if_snd* queue, thus transmitting all interactive packets before any other packets.

There are no standard SNMP variables to count packets queued according to the TOS fields. The XXX comment in line 353 indicates that the SLIP driver is counting low-delay packets in *if_omcasts*, *not* multicast packets.

359-383 If the SLIP interface is tapped by BPF, slstart makes a copy of the output packet before any header compression occurs. The copy is saved on the stack in the *bpfbuf* array.

384-388 If compression is enabled and the packet contains a TCP segment, sloutput calls *sl_compress_tcp*, which attempts to compress the packet. The resulting packet type is returned and logically ORed with the first byte in IP header (Section 29.13).

389-398 The compressed header is now copied into the BPF header, and the direction recorded as *SLIPDIR_OUT*. The completed BPF packet is passed to *bpf_tap*.

483-484 slstart returns if the for loop terminates.

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_oproc) (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
```

— if_sl.c

```

370      * this is a serial line, packets should be short
371      * and/or the copy should be negligible cost compared
372      * to the packet transmission time).
373      */
374      struct mbuf *ml = m;
375      u_char *cp = bpdbuf + SLIP_HDRLEN;
376      len = 0;
377      do {
378          int      mlen = ml->m_len;
379          bcopy(mtod(ml, caddr_t), cp, mlen);
380          cp += mlen;
381          len += mlen;
382      } while (ml = ml->m_next);
383  }
384  if ((ip = mtod(m, struct ip *))>ip_p == IPPROTO_TCP) {
385      if (sc->sc_if.if_flags & SC_COMPRESS)
386          *mtod(m, u_char *) |= sl_compress_tcp(m, ip,
387                                              &sc->sc_comp, 1);
388  }
389  if (sc->sc_bpf) {
390      /*
391      * Put the SLIP pseudo-"link header" in place.  The
392      * compressed header is now at the beginning of the
393      * mbuf.
394      */
395      bpdbuf[SLX_DIR] = SLIPDIR_OUT;
396      bcopy(mtod(m, caddr_t), &bpdbuf[SLX_CHDR], CHDR_LEN);
397      bpf_tap(sc->sc_bpf, bpdbuf, len + SLIP_HDRLEN);
398  }

```

/* packet output code */

```

483 }
484 }
```

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

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.

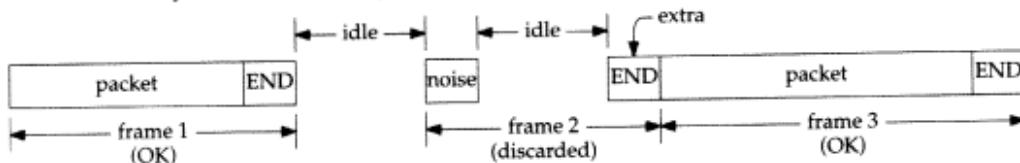


Figure 5.18 Karn's method for discarding noise on a SLIP line.

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

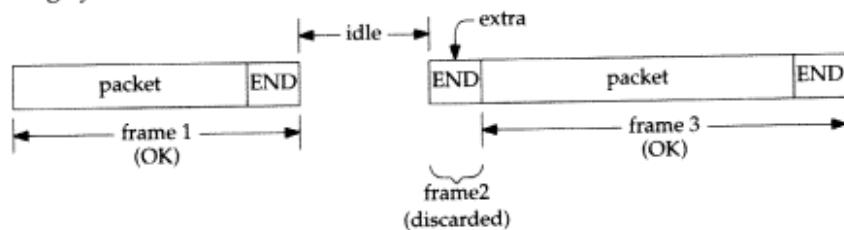


Figure 5.19 Karn's method with no noise.

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

```

if_sl.c                                     if_sl.c
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  }

```

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.

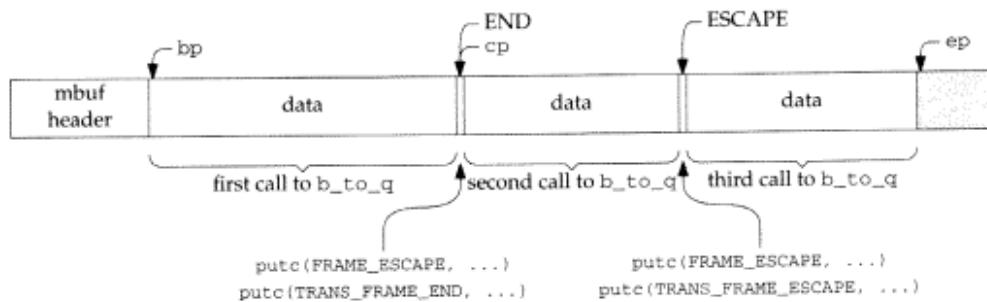


Figure 5.21 SLIP transmission of a single mbuf.

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)) {
469     /*
470      * Not enough room. Remove a char to make room
471      * and end the packet normally.
472      * If you get many collisions (more than one or two
473      * a day) you probably do not have enough clists
474      * and you should increase "nclist" in param.c.
475      */
476     (void) unputc(&tp->t_outq);
477     (void) putc(FRAME_END, &tp->t_outq);
478     sc->sc_if.if_collisions++;
479   } else {
480     ++sc->sc_if.if_oBYTES;
481     sc->sc_if.if_opackets++;
482   }

```

if_sl.c

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 *slistart* 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     ttywflush(tp);
217     s = splimp();           /* actually, max(spltty, splnet) */
218     tp->t_line = 0;
219     sc = (struct sl_softc *) tp->t_sc;
220     if (sc != NULL) {
221         if_down(&sc->sc_if);
222         sc->sc_ttyp = NULL;
223         tp->t_sc = NULL;
224         MCLFREE((caddr_t) (sc->sc_ep - SLBUFSIZE));
225         sc->sc_ep = 0;
226         sc->sc_mp = 0;
227         sc->sc_buf = 0;
228     }
229     splx(s);
230 }
```

if_sl.c

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 `sliioctl` processes `ioctl` commands issued for a SLIP interface through a socket descriptor. In Section 4.4 we showed how `ifioctl` calls `sliioctl`. 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
<code>SLIOCGUNIT</code>	<code>int *</code>	<code>sltioctl</code>	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.

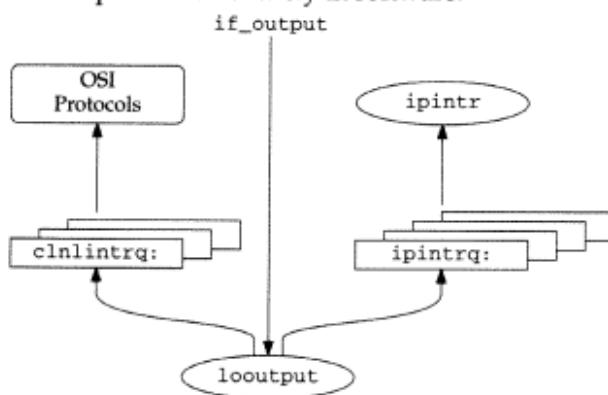


Figure 5.26 Loopback device driver.

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

```

57 int                                         if_loop.c
58 looutput(ifp, m, dst, rt)
59 struct ifnet *ifp;
60 struct mbuf *m;
61 struct sockaddr *dst;
62 struct rtentry *rt;
63 {
64     int      s, isr;
65     struct ifqueue *ifq = 0;
66     if ((m->m_flags & M_PKTHDR) == 0)
67         panic("looutput no HDR");
68     ifp->if_lastchange = time;
69     if (loif.if_bpf) {
70         /*
71          * We need to prepend the address family as
72          * a four byte field.  Cons up a dummy header

```

```
structure
queued for
outgoing
destination
of an out-
we'll see
shown in
--- if_loop.c

    * to pacify bpf. This is safe because bpf
    * will only read from the mbuf (i.e., it won't
    * try to free it or keep a pointer to it).
    */
    struct mbuf m0;
    u_int af = dst->sa_family;

    m0.m_next = m;
    m0.m_len = 4;
    m0.m_data = (char *) &af;

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

if (rt && rt->rt_flags & (RTF_REJECT | RTF_BLACKHOLE)) {
    m_freem(m);
    return (rt->rt_flags & RTF_BLACKHOLE ? 0 :
           rt->rt_flags & RTF_HOST ? EHOSTUNREACH : ENETUNREACH);
}
ifp->if_opackets++;
ifp->if_obytes += m->m_pkthdr.len;
switch (dst->sa_family) {
case AF_INET:
    ifq = &ipintrq;
    isr = NETISR_IP;
    break;

case AF_ISO:
    ifq = &clnlntrq;
    isr = NETISR_ISO;
    break;

default:
    printf("lo%d: can't handle af%d\n", ifp->if_unit,
           dst->sa_family);
    m_freem(m);
    return (EAFNOSUPPORT);
}
s = splimp();
if (IF_QFULL(ifq)) {
    IF_DROP(ifq);
    m_freem(m);
    splx(s);
    return (ENOBUFS);
}
IF_ENQUEUE(ifq, m);
schednetisr(isr);
ifp->if_ipackets++;
ifp->if_obytes += m->m_pkthdr.len;
splx(s);
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.

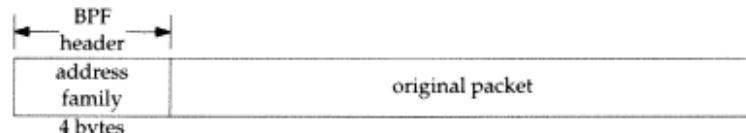


Figure 5.28 BPF loopback packet: logical format.

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.

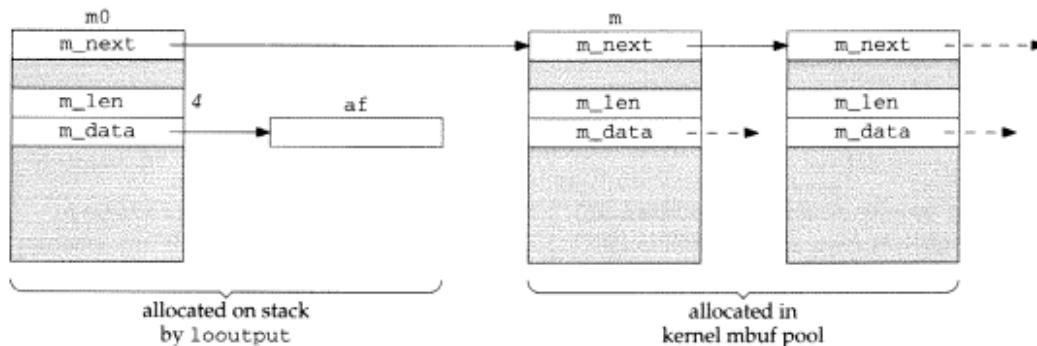


Figure 5.29 BPF loopback packet: mbuf format.

`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 schednetisr.

5.5 Summary

We described the two remaining interfaces to which we refer throughout the text: s10, 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 s10input 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	
B	128.0.0.0 to 191.255.255.255	unicast
C	192.0.0.0 to 223.255.255.255	
D	224.0.0.0 to 239.255.255.255	multicast
E	240.0.0.0 to 247.255.255.255	experimental

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).

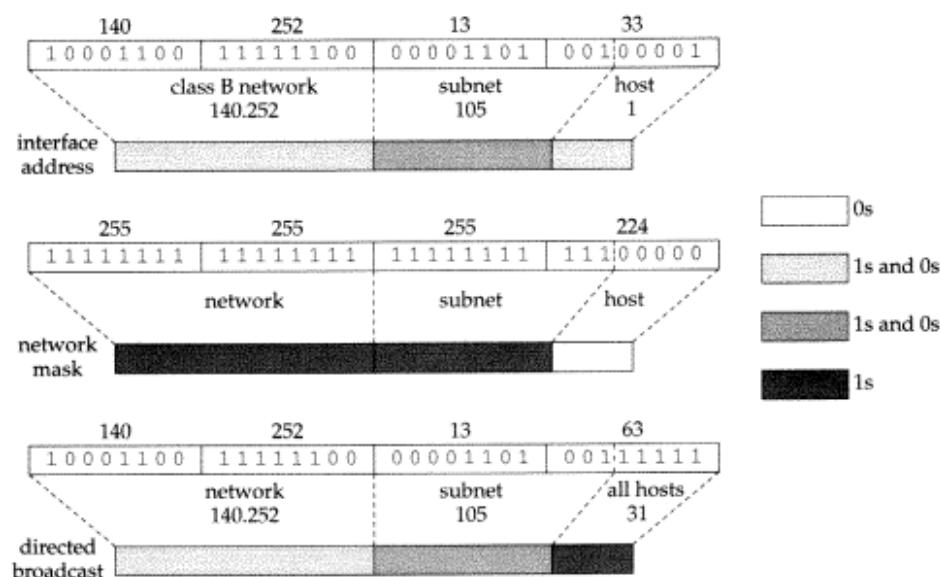


Figure 6.2 Alternate IP address notations.

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

now more
as an elec-
neter and
d all code
Net/3 ker-

itions and

ed in this

SLIP inter-
de in their
interface
address.
the link-

from Fig-
e list con-

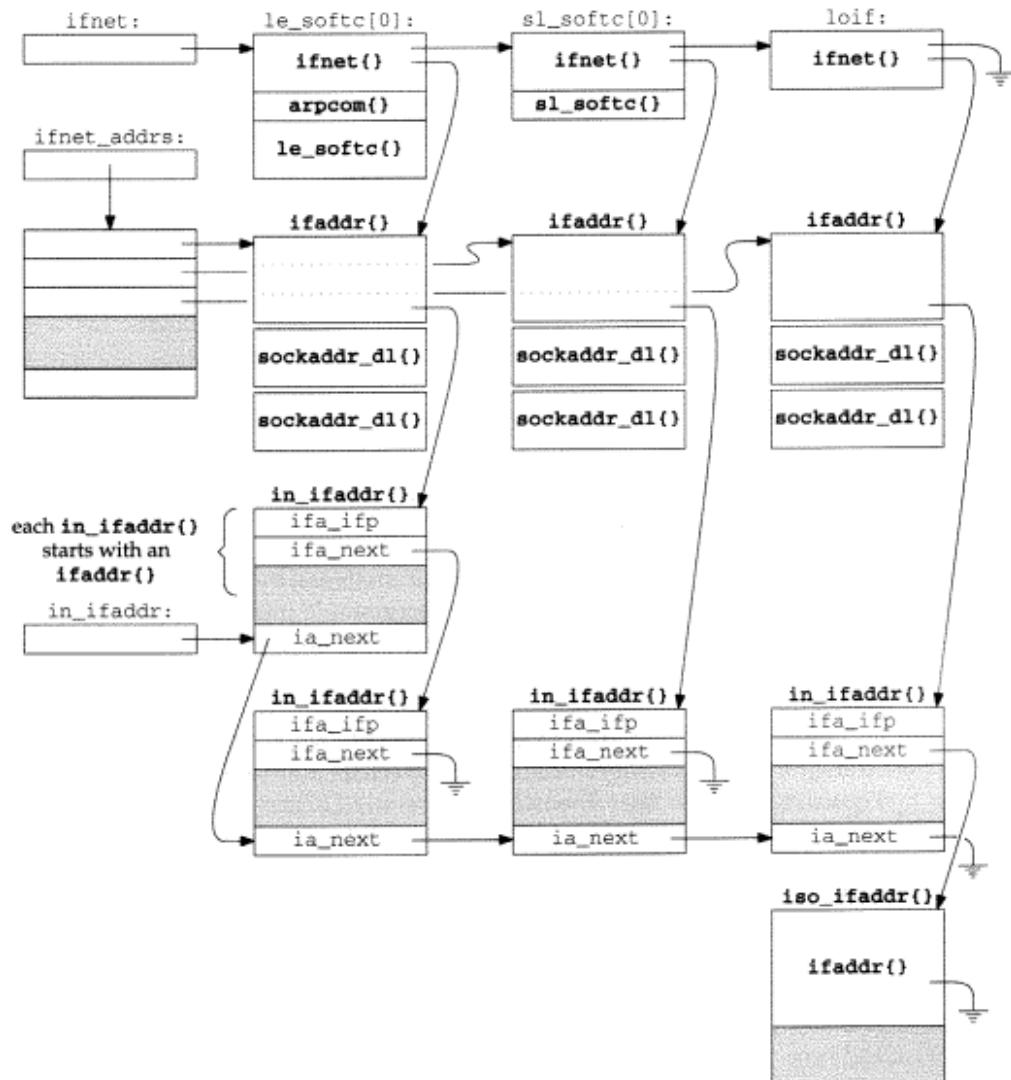


Figure 6.5 Interface and address data structures.

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:

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

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 };
```

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.

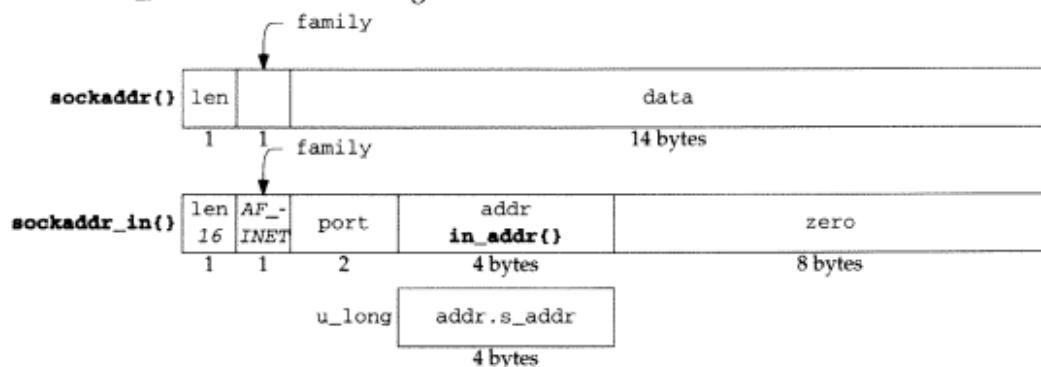


Figure 6.7 The organization of a sockaddr_in structure (sin_zero omitted).

Usually, when an Internet addresses 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 };

```

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
SIOCGIFADDR	struct ifreq *	in_control	get interface address
SIOCGIFNETMASK	struct ifreq *	in_control	get interface netmask
SIOCGIFDSTADDR	struct ifreq *	in_control	get interface destination address
SIOCIFBRDADDR	struct ifreq *	in_control	get interface broadcast address
SIOCSIFADDR	struct ifreq *	in_control	set interface address
SIOCSIFNETMASK	struct ifreq *	in_control	set interface netmask
SIOCSIFDSTADDR	struct ifreq *	in_control	set interface destination address
SIOCIFBRDADDR	struct ifreq *	in_control	set interface broadcast address
SIOCDIFADDR	struct ifreq *	in_control	delete interface address
STOCAIFADDR	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.

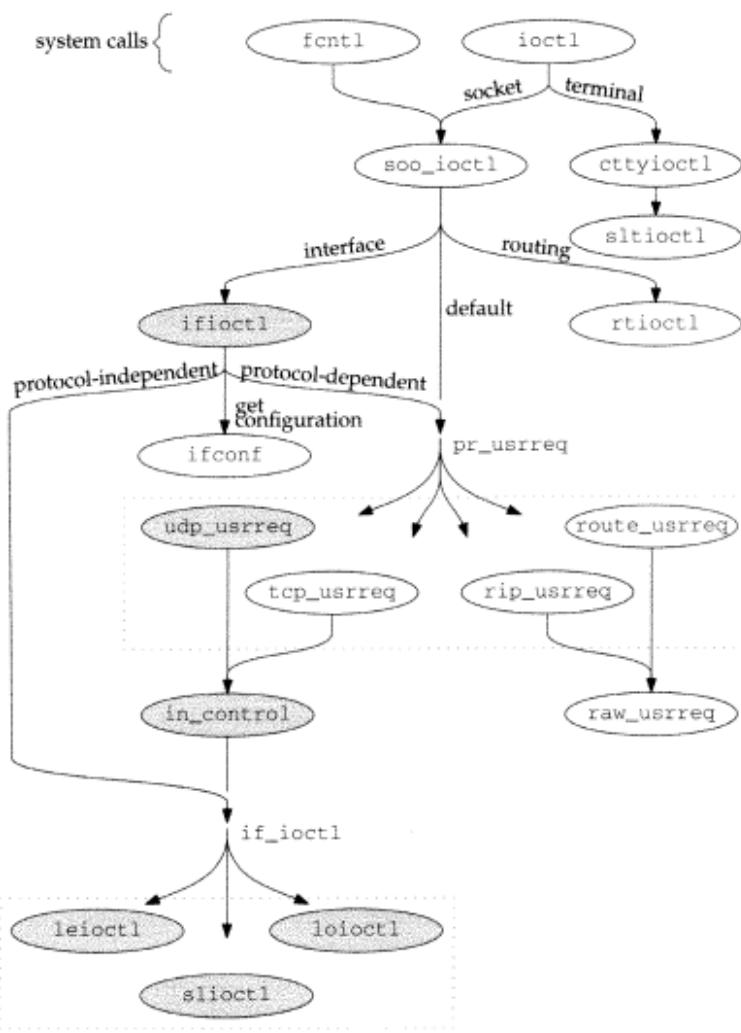


Figure 6.11 ioctl functions described in this chapter.

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.

```
if.c
447     default:
448         if (so->so_proto == 0)
449             return (EOPNOTSUPP);
450         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
451                                         cmd, data, ifp));
452     }
453     return (0);
454 }
```

if.c

if.c

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 sooo_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 sooo_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.

```

in.c
132 in_control(so, cmd, data, ifp)
133 struct socket *so;
134 int     cmd;
135 caddr_t data;
136 struct ifnet *ifp;
137 {
138     struct ifreq *ifr = (struct ifreq *) data;
139     struct in_ifaddr *ia = 0;
140     struct ifaddr *ifa;
141     struct in_ifaddr *oia;
142     struct in_aliasreq *ifra = (struct in_aliasreq *) data;
143     struct sockaddr_in oldaddr;
144     int      error, hostIsNew, maskIsNew;
145     u_long   i;

146     /*
147      * Find address for this interface, if it exists.
148      */
149     if (ifp)
150         for (ia = in_ifaddr; ia; ia = ia->ia_next)
151             if (ia->ia_ifp == ifp)
152                 break;

153     switch (cmd) {

                /* 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 }

```

in.c

Figure 6.13 in_control function.

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;

```

in.c

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.

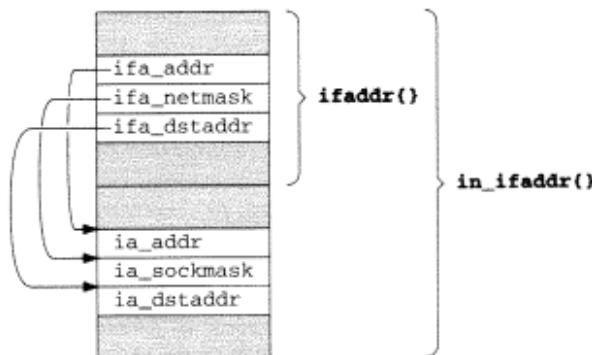


Figure 6.15 An in_ifaddr structure after initialization by in_control.

```

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

*in.c**in.c*

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, sliioctl, and loiioctl 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 = ntohs(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     }

```

*in.c*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.

in.c

```

404     /*
405      * Add route for the network.
406      */
407     ia->ia_ifa.ifa_metric = ifp->if_metric;
408     if (ifp->if_flags & IFF_BROADCAST) {
409         ia->ia_broadaddr.sin_addr.s_addr =
410             htonl(ia->ia_subnet | ~ia->ia_subnetmask);
411         ia->ia_netbroadcast.s_addr =
412             htonl(ia->ia_net | ~ia->ia_netmask);
413     } else if (ifp->if_flags & IFF_LOOPBACK) {
414         ia->ia_ifa.ifa_dstaddr = ia->ia_ifa.ifa_addr;
415         flags |= RTF_HOST;
416     } else if (ifp->if_flags & IFF_POINTOPOINT) {
417         if (ia->ia_dstaddr.sin_family != AF_INET)
418             return (0);
419         flags |= RTF_HOST;
420     }
421     if ((error = rtinit(&(ia->ia_ifa), (int) RTM_ADD, flags)) == 0)
422         ia->ia_flags |= IFA_ROUTE;
423     /*
424      * If the interface supports multicast, join the "all hosts"
425      * multicast group on that interface.
426      */
427     if (ifp->if_flags & IFF_MULTICAST) {
428         struct in_addr addr;
429
430         addr.s_addr = htonl(INADDR_ALLHOSTS_GROUP);
431         in_addmulti(&addr, ifp);
432     }
433 }
```

*in.c**in.c*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.

