Principia Softwarica: The Plan 9 Network Stack `/net/`

version 0.1

Yoann Padioleau
`yoann.padioleau@gmail.com`

with code from
Dave Presotto and Phil Winterbottom

April 1, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2021 Plan 9 Foundation.
MIT license.

Contents

1 Introduction **10**
1.1 Motivations . . . 10
1.2 The Plan 9 network stack: `/net` . . . 10
1.3 Other network stacks . . . 11
1.4 Getting started . . . 11
1.5 Requirements . . . 13
1.6 About this document . . . 13
1.7 Copyright . . . 13
1.8 Acknowledgments . . . 13

2 Overview **14**
2.1 Networking principles . . . 14
2.1.1 Local area networking . . . 14
2.1.2 Packets and frames . . . 14
2.1.3 Internetworking and gateways . . . 14
2.1.4 Reliable communication . . . 15
2.1.5 Network stack . . . 15
2.1.6 Layers . . . 15
2.2 `/net/` files interface . . . 15
2.3 `helloclient.c` and `helloserver.c` . . . 15
2.4 Code organization . . . 17
2.5 Software architecture . . . 17
2.5.1 Trace of a network write . . . 18
2.5.2 Trace of a network read . . . 18
2.6 Book structure . . . 18

3 Core Data Structures **20**
3.1 IP addresses . . . 20
3.1.1 IPv4 vs IPv6 . . . 20
3.1.2 Network class and network identifier . . . 22
3.1.3 Network mask . . . 22
3.1.4 Network byte order . . . 23
3.1.5 Other address operations . . . 23
3.2 IP header . . . 23
3.3 IP Interfaces: `Ipifc` . . . 24
3.3.1 User side . . . 24
3.3.2 Kernel side . . . 25
3.4 Physical link . . . 26
3.4.1 `Medium` . . . 26
3.4.2 `media` . . . 27

3.4.3 Ethernet medium . . . 27
3.4.4 Ethernet controllers and etherxx . . . 28
3.5 /net filesystem . . . 29
3.5.1 Fs and ipfs . . . 29
3.5.2 IP fragments and IP statistics . . . 30
3.5.3 Protocols . . . 32
3.5.4 Conversations . . . 33
3.5.5 Chan Qid, PROTO(), CONV(), QID() . . . 34
3.6 /net/ether filesystem . . . 35
3.6.1 Netif . . . 35
3.6.2 Netfile . . . 36
3.7 Routes . . . 37
3.7.1 Single Route . . . 37
3.7.2 Routing forest . . . 38
3.8 Blocks . . . 39

4 Initialization **40**
4.1 Kernel side . . . 40
4.1.1 Mounting the ip device . . . 40
4.1.2 Mounting the ethernet device . . . 42
4.2 User side . . . 43
4.2.1 Connect manually . . . 43
4.2.2 dial() . . . 43

5 User/Kernel Bridge **44**
5.1 IP device . . . 44
5.1.1 /net hierarchy and ipwalk() . . . 45
5.1.2 Dispatch functions, ipxxx() . . . 48
5.1.3 /net/x/clone . . . 52
5.1.4 /net/x/y/ctl . . . 54
5.1.5 /net/x/y/data . . . 61
5.1.6 /net/x/y/err . . . 62
5.1.7 /net/x/y/listen . . . 62
5.1.8 Other files . . . 63
5.2 Ethernet device . . . 65
5.2.1 /net/etherx hierarchy and etherwalk() . . . 66

6 Configuration **67**
6.1 /net/ipifc/ protocol . . . 67
6.1.1 Protocol initialisation . . . 67
6.1.2 /net/ipifc/clone . . . 68
6.1.3 Binding medium: /net/ipifc/x/ctl bind . . . 69
6.1.4 /net/ipifc/x/ctl . . . 70
6.1.5 Adding an IP: /net/ipifc/x/ctl add . . . 71
6.2 IP Interface, user side . . . 73
6.2.1 Parsing . . . 73
6.2.2 /net/ipifc/x . . . 76
6.2.3 /net/ipifc/stats . . . 77
6.3 Binding ethernet medium . . . 77
6.3.1 etherbind() . . . 77

6.3.2 `chandial()` . . . 79
6.4 Setting up routes . . . 79
6.5 Advanced configurations . . . 79
6.5.1 IP Routing . . . 79
6.5.2 MTU . . . 80
6.5.3 TOS . . . 81
6.5.4 TTL . . . 81
6.5.5 Dynamic interface, removing, unbinding, etc. . . . 82
6.5.6 Unbind on close . . . 84

7 Physical Transport: Ethernet 85
7.1 Ethernet addresses . . . 85
7.1.1 Parsing . . . 85
7.1.2 `/net/etherx/addr` . . . 85
7.2 Ethernet header . . . 86
7.3 Ethernet packet . . . 86
7.4 IO . . . 87
7.4.1 Writing . . . 87
7.4.2 Reading . . . 87
7.5 Advanced features . . . 87

8 Inter Network Transport: IP 88
8.1 IP addresses . . . 88
8.1.1 Parsing . . . 88
8.1.2 Comparisons . . . 90
8.2 IP header . . . 90
8.2.1 Byte ordering . . . 90
8.2.2 Checksum . . . 90
8.3 IP Fragments . . . 92
8.4 IO . . . 93
8.4.1 Writing: `ipoput4()` . . . 93
8.4.2 Reading: `ipiput4()` . . . 97
8.5 Advanced features . . . 103
8.5.1 Gating . . . 103
8.5.2 Manual fragmentation setting . . . 104
8.5.3 Routing . . . 104
8.5.4 Reassembling . . . 105

9 Finding Machines Locally: ARP 106
9.1 Initialisation . . . 107
9.2 `/net/arp` . . . 108
9.2.1 Reading . . . 108
9.2.2 Writing . . . 109

10 Finding Machines Globally: Routes 112
10.1 `v4lookup()` . . . 112
10.2 `findipifc()` . . . 113
10.3 Adding routes: `v4addroute()` . . . 114
10.4 Broadcast routes . . . 115
10.5 Route managment . . . 115

10.5.1 Allocation . . . 115
10.5.2 Free . . . 116
10.5.3 Insertion . . . 116
10.5.4 Balancing . . . 119
10.6 /net/iproutes . . . 120
10.6.1 Reading . . . 120
10.6.2 Writing . . . 123
10.7 Self cache . . . 125
10.7.1 Data structures . . . 126
10.7.2 Adding IPs . . . 127
10.7.3 Removing IPs . . . 128
10.7.4 /net/ipselftab . . . 130
10.8 Advanced features . . . 131
10.8.1 Removing routes . . . 131
10.8.2 Flushing routes . . . 132
10.8.3 Router boards . . . 133

11 Basic Communication: UDP **134**
11.1 Protocol initialisation . . . 134
11.2 Protocol data structures . . . 135
11.2.1 Udppriv . . . 135
11.2.2 Addresses hashtable: Ipht . . . 135
11.2.3 Statistics . . . 137
11.3 Protocol header . . . 137
11.4 /net/tcp/clone . . . 138
11.5 /net/tcp/x/ctl . . . 139
11.5.1 Connect . . . 139
11.5.2 Announce . . . 140
11.6 IO . . . 140
11.6.1 Writing: udpkick() . . . 140
11.6.2 Reading: udpiput() . . . 141
11.7 Other files . . . 147
11.7.1 /net/tcp/stats . . . 147
11.7.2 /net/tcp/x/status . . . 147
11.8 Advanced features . . . 147
11.8.1 Ignore advice . . . 147
11.8.2 Not regular forme . . . 148
11.8.3 User level UDP headers . . . 148

12 Reliable Communication: IL **150**
12.1 Protocol initialisation . . . 150
12.2 Protocol data structures . . . 151
12.2.1 IL state . . . 152
12.2.2 IL control packet type . . . 152
12.2.3 Ilcb . . . 153
12.2.4 Ilpriv . . . 154
12.2.5 Statistics . . . 154
12.3 Protocol header . . . 155
12.4 /net/il/clone . . . 155
12.5 /net/il/x/ctl . . . 156

12.5.1 Connect . . . 156
12.5.2 Announce . . . 156
12.6 IO . . . 157
12.6.1 Writing: `ilkick()` . . . 157
12.6.2 Reading: `iliput()` . . . 158
12.7 State machine . . . 160
12.7.1 `ilsendctl()` . . . 160
12.7.2 `ilprocess()` . . . 162
12.7.3 Start . . . 165
12.7.4 Close . . . 166
12.7.5 Syncer . . . 167
12.7.6 Syncee . . . 167
12.7.7 Establised . . . 167
12.8 Features . . . 167
12.8.1 Reliable datagram service . . . 167
12.8.2 In sequence delivery . . . 168
12.8.3 Retransmission of lost messages . . . 168
12.8.4 Out-of-order messages saving . . . 168
12.8.5 Adaptive timeouts . . . 168
12.8.6 Keep-alive . . . 168
12.9 Other files . . . 168
12.9.1 `/net/il/stats` . . . 168
12.9.2 `/net/il/x/status` . . . 168
12.10 Advanced features . . . 168
12.10.1 Fast timeout . . . 168
12.10.2 Special extension . . . 169

13 Standard Reliable Communication: TCP **170**

14 Applications **171**
14.1 Remote shell: Telnet . . . 171
14.2 Remote file access: TFTP . . . 171
14.3 Mail . . . 171

15 Remote Procedure Call: 9P **172**

16 Network File System: `exportfs` **173**

17 Name Resolution: DNS **174**

18 Security **175**
18.1 Denial of service . . . 175

19 Debugging Support **176**

20 Profiling Support **177**

21 Advanced Topics **178**
21.1 Flow Control . . . 178
21.2 IPv6 . . . 178
21.3 Broadcast . . . 178

21.4 Multicast . . . 179
21.5 Network database: `ndb` . . . 181
21.5.1 `/net/ndb` . . . 181
21.6 Connection Server . . . 182
21.7 Sniffing . . . 182
21.8 Point to point interface . . . 182
21.9 Proxy . . . 182
21.10 Wifi . . . 183
21.11 VLAN . . . 183
21.12 VPN . . . 183
21.13 Packet filter . . . 183

22 Conclusion 184
22.1 Patterns and techniques . . . 184
22.2 Connections to other books . . . 184
22.3 Beyond the Plan 9 network stack . . . 185

A Debugging 186
A.1 Dumpers . . . 186
A.1.1 Addresses . . . 186
A.1.2 IP interface . . . 188
A.2 `/net/log` . . . 188
A.2.1 `Netlog` . . . 188
A.2.2 `/net/log` . . . 189
A.2.3 Opening . . . 190
A.2.4 Reading . . . 190
A.2.5 Control . . . 191
A.3 `/bin/snoopy` . . . 192
A.3.1 `/net/ipifc/x/snoop` . . . 192
A.3.2 `/bin/snoopy` . . . 193

B Profiling 194
B.1 `/net/x/stats` . . . 194

C Error Management 195

D Ethernet NE2000 Driver 196
D.1 Data structures . . . 196
D.1.1 Ethernet `Dp8390` controller . . . 196
D.2 Initialisation . . . 197
D.3 Mounting . . . 201
D.4 IO . . . 202
D.4.1 Writing . . . 202
D.4.2 Reading . . . 205
D.5 Advanced features . . . 210
D.5.1 Multicast . . . 210
D.5.2 Promiscuous mode . . . 212
D.5.3 Power managment . . . 212
D.5.4 Plug and play . . . 212

E Utilities **215**
E.1 Globbing . . . 215

F Extra Mediums **216**
F.1 Null medium . . . 216
F.2 Loopback medium . . . 217
F.3 Point to point serial line . . . 219
F.4 Token ring . . . 219

G Extra Protocols **220**
G.1 ICMP . . . 220
G.2 RUDP . . . 220
G.3 GRE . . . 220
G.4 ESP . . . 220
G.5 Datakit and URP . . . 220

H Extra Applications **221**
H.1 Remote login, rlogin . . . 221
H.2 Serving files, FTP . . . 221
H.3 Serving documents, HTTP . . . 221
H.4 Serving mails, SMTP . . . 221
H.5 Serving news, NNTP . . . 221

I Extra Code **222**
I.1 `include/net/` . . . 222
I.1.1 `include/net/ip.h` . . . 222
I.2 `libip/` . . . 224
I.2.1 `libip/parseether.c` . . . 224
I.2.2 `libip/parseip.c` . . . 224
I.2.3 `libip/myetheraddr.c` . . . 225
I.2.4 `libip/myipaddr.c` . . . 225
I.2.5 `libip/eipfmt.c` . . . 226
I.2.6 `libip/equivip.c` . . . 227
I.2.7 `libip/ipaux.c` . . . 227
I.2.8 `libip/bo.c` . . . 228
I.2.9 `libip/classmask.c` . . . 229
I.2.10 `libip/ptclbsum.c` . . . 230
I.2.11 `libip/readipifc.c` . . . 231
I.3 `libip/tests/` . . . 231
I.3.1 `libip/tests/testreadipifc.c` . . . 231
I.4 `kernel/network/ip/` . . . 232
I.4.1 `kernel/network/ip/ip.h` . . . 232
I.4.2 `kernel/network/ip/ip.c` . . . 238
I.5 `kernel/network/` . . . 239
I.5.1 `kernel/network/portfns_network.h` . . . 239
I.5.2 `kernel/network/netif.h` . . . 240
I.5.3 `kernel/network/netif.c` . . . 241
I.6 `kernel/network/386/` . . . 255
I.6.1 `kernel/network/etherif.h` . . . 255
I.6.2 `kernel/network/386/devether.c` . . . 255

I.6.3 kernel/network/386/ether8390.h . . . 265
I.6.4 kernel/network/386/ether8390.c . . . 266
I.6.5 kernel/network/386/ether2000.c . . . 270
I.6.6 kernel/network/386/etherigbe.c . . . 271
I.6.7 kernel/network/386/ethermii.c . . . 310
I.6.8 kernel/network/386/ethermii.h . . . 314
I.7 kernel/network/arm/ . . . 317
I.7.1 network/arm/devether.c . . . 317
I.8 kernel/network/ip/ . . . 327
I.8.1 kernel/network/ip/devip.c . . . 327
I.8.2 kernel/network/ip/arp.c . . . 333
I.8.3 kernel/network/ip/icmp.c . . . 342
I.8.4 kernel/network/ip/chandial.c . . . 353
I.8.5 kernel/network/ip/nullmedium.c . . . 355
I.8.6 kernel/network/ip/loopbackmedium.c . . . 356
I.8.7 kernel/network/ip/ethermedium.c . . . 356
I.8.8 kernel/network/ip/pktmedium.c . . . 367
I.8.9 kernel/network/ip/iproute.c . . . 369
I.8.10 kernel/network/ip/ipaux.c . . . 371
I.8.11 kernel/network/ip/ipifc.c . . . 374
I.8.12 kernel/network/ip/netlog.c . . . 388
I.8.13 kernel/network/ip/ptclbsum.c . . . 390
I.8.14 kernel/network/ip/il.c . . . 391
I.8.15 kernel/network/ip/udp.c . . . 403
I.8.16 kernel/network/ip/tcp.c . . . 406
I.9 kernel/network/ip/ ipv6 . . . 468

Glossary **520**

Index **521**

References **567**

Chapter 1

Introduction

The goal of this book is to present with full details the source code of a network stack.

1.1 Motivations

Why a network stack? Because I think you are a better programmer if you fully understand how things work under the hood.

Nowadays, nearly every program communicates over a network. Web browsers, databases, chat applications, cloud services—they all depend on the network stack to deliver data reliably between machines. Yet most programmers treat the network as a black box: they call `dial()` or `connect()` and trust that packets will arrive. Understanding the internals of a network stack removes that mystery and gives you a much better mental model for diagnosing network problems, writing performant servers, and understanding protocols.

Here are a few questions I hope this book will answer:

- What happens when you open a connection to another machine?
- How are opened connections stored in the kernel?
- How can the kernel dispatch incoming packets to the right process?
- How can the network stack offer reliable communications over an unreliable physical network?

1.2 The Plan 9 network stack: `/net`

I will explain in this book the code of the Plan 9 network stack, which is implemented entirely in the kernel behind the `/net/` file interface. The stack includes Ethernet, IP, ARP, routing, UDP, IL (Plan 9's own reliable transport protocol), and TCP. In total, the networking code is roughly 10 000 lines of C. Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. The Plan 9 network stack has an unusual and elegant property: every network connection is a directory of files. Opening a connection means creating a directory under `/net/tcp/` or `/net/udp/`, and communicating means reading and writing files in that directory. There are no sockets, no `ioctl`—just the same `open`/`read`/`write` interface used for everything else in Plan 9. Another advantage is that Plan 9 includes IL (Internet Link), a reliable transport protocol simpler than TCP. IL conveys the essential ideas of reliable communication—sequence numbers, acknowledgments, retransmission—without TCP's complexity (window scaling, congestion control, urgent data, etc.), making it an ideal teaching vehicle.

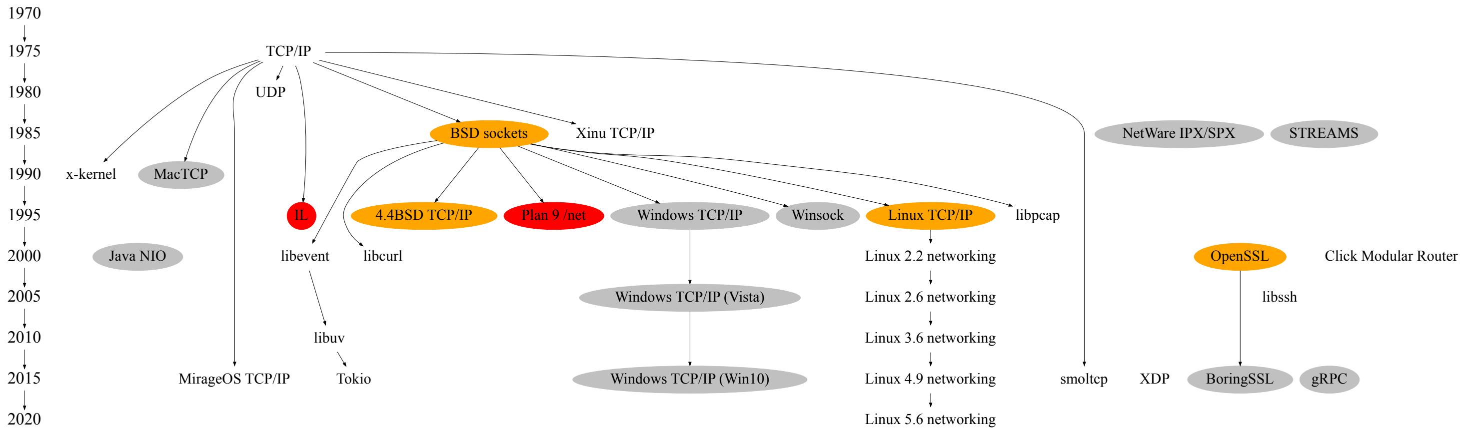

Figure 1.1: Network stacks timeline

1.3 Other network stacks

Here are a few network stacks that I considered for this book, but which I ultimately discarded:

- DataKit was Bell Labs' packet-switching network, a precursor to the internet within AT&T. Its ideas influenced Plan 9's networking, but DataKit itself is no longer in use.
- XINU's network stack is educational but too simple—it lacks features like routing and a complete TCP implementation.
- The Linux network stack is production-quality and heavily optimized, but enormous. Even the core IP/TCP code (excluding drivers) is hundreds of thousands of lines.
- The BSD network stack (especially FreeBSD's) has an excellent reputation and was the reference implementation of TCP/IP for decades. Windows even adopted a fork of the BSD stack at one point. However, like Linux, it is far too large for a book.
- Minix has a network stack, but it runs as a separate user-space server (microkernel architecture), making it harder to compare with Plan 9's in-kernel approach.
- Mirage[1] is a TCP/IP stack written in OCaml as a unikernel library. An interesting modern approach, but the OCaml code is harder to relate to the C-based stacks most programmers encounter.

Figure 1.1 presents a timeline of major network stack implementations. I think the Plan 9 network stack represents the best compromise for this book: it implements the essential protocols (Ethernet, IP, ARP, UDP, IL, TCP) in a clean, readable codebase, with the added elegance of the `/net/` file interface.

1.4 Getting started

To test the network stack, you need Plan 9 running under QEMU with networking enabled:

```
$ qemu-system-i386 -smp 4 -m 512 -kernel $KERNEL \
    -hda dosdisk.img -serial mon:stdio -net nic -net user
```

The `-net nic -net user` flags tell QEMU to emulate a network card and provide a virtual network where the guest (Plan 9) gets IP address `10.0.2.15`, the gateway is `10.0.2.2`, and a DNS server is at `10.0.2.3`.

The quickest way to get networking up inside Plan 9 is:

[1] https://github.com/mirage/mirage-tcpip

```
$ ipconfig
$ dns -r
$ cs
$ hget http://www.google.com
```

`ipconfig` obtains an IP address via DHCP. `dns -r` starts a DNS resolver. `cs` starts the connection server, which translates network addresses for programs like `hget`. Finally, `hget` fetches a web page—if this works, the entire stack (Ethernet, IP, TCP, DNS) is functional.

To explore the network state in more detail, you can inspect the `/net/` files directly:

```
$ ipconfig
$ cat /net/ipifc/0/status
device /net/ether0 ... 10.0.2.15 /120 10.0.2.0
$ ping 10.0.2.15
0: 2900microsec
1: ...
$ ping 10.0.2.2
...
$ cat /net/iproute
0.0.0.0 /96 10.0.2.2 4 none -
...
$ cat /net/arp
...
$ cat /net/ndb
...
```

`/net/ipifc/0/status` shows the network interface configuration: which Ethernet device it is bound to and which IP address was assigned. `ping` tests connectivity by sending ICMP packets. `/net/iproute` displays the routing table, `/net/arp` shows the ARP cache (MAC-to-IP mappings learned from the local network), and `/net/ndb` contains the network database (DNS server address, etc.). All of these are just files you can `cat`—no special diagnostic commands needed.

DNS resolution can be tested interactively with `dnsquery`:

```
$ dns -r
 starting dns resolver on 10.0.2.15's /net
$ dnsquery
> www.google.com
www.google.com ip 216.58.204.132
```

You can even make raw TCP connections with `telnet` using Plan 9's network address syntax `proto!host!port`:

```
$ telnet tcp!216.58.204.132!80
GET
```

This opens a TCP connection to Google's IP on port 80 and sends a bare HTTP `GET` request—you should see HTML content in response.

For a higher-level test, `hget` fetches a URL. It requires the connection server `cs` to translate hostnames:

```
$ cs
$ hget http://www.google.com
```

Finally, for a full web browsing experience under Plan 9, you need several cooperating file servers:

```
$ plumber
$ webcookies
$ webfs
$ abaco
```

`plumber` handles inter-application message routing (mounted on `/mnt/plumb`), `webcookies` manages HTTP cookies (mounted on `/mnt/webcookies`), `webfs` provides a file-system interface to the web (mounted on `/mnt/web`), and `abaco` is the actual web browser that reads from `webfs`. This is the Plan 9 philosophy in action: each service is a separate file server, composed together.

Note that the network stack described in this book is implemented entirely inside the Plan 9 kernel and can only run under a full Plan 9 system—it is not available through plan9port. The user-space networking functions (`dial()`, `announce()`, `listen()`) do exist in plan9port, but they are thin wrappers around the host OS's socket interface, not the Plan 9`/net/` file system described here.

1.5 Requirements

Because this book is made of C source code, you will need a good knowledge of the C programming language [KR88]. The networking code makes heavy use of byte-level manipulation (packing and unpacking packet headers) and concurrent programming (kernel processes handling incoming packets). This book is not an introduction to computer networking. I assume you have a basic understanding of concepts such as IP addresses, ports, TCP, UDP, Ethernet, and the layered protocol model. Standard references include Tanenbaum's *Computer Networks* [TW10], Comer's *Internetworking with TCP/IP* [Com13], and Stevens' *TCP/IP Illustrated* [Ste94] (for protocol internals) and *UNIX Network Programming* [Ste03] (for the programming interface). Reading the Kernel book [Pad14] is strongly recommended: the network stack runs inside the kernel and uses its facilities (devices, `sleep`/`wakeup`, kernel processes). The historical paper *The Organization of Networks in Plan 9* [?] is also a useful reference for the design decisions behind `/net/`.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

Most of this document is made of source code from Plan 9, so those parts are copyright by Plan 9 Foundation. The prose is mine and is licensed under the All Rights Reserved License.

1.8 Acknowledgments

I would like to acknowledge of course the Plan9's authors who wrote most of this book: Dave Presotto, Phil Winterbottom, and many other people from Bell Labs.

Chapter 2

Overview

Before showing the source code of the Plan 9 network stack in the following chapters, I first give an overview of the general principles of networking, describe the `/net/` file interface, and explain how the code is organized.

2.1 Networking principles

Networking solves a fundamental problem: how to let two programs on different machines communicate. The solution involves multiple layers of abstraction, from the physical signals on a wire to the reliable byte streams that applications use. The following subsections present these layers bottom-up, from local networks to reliable transport—the same order in which the Plan 9 network stack processes packets.

2.1.1 Local area networking

A local area network (LAN) connects machines that are physically close—in the same room, building, or campus. The dominant LAN technology is Ethernet, where each machine has a unique hardware address (the MAC address) and sends data as frames on a shared medium. Every machine on the LAN sees every frame, but only the one with the matching destination MAC address accepts it.

2.1.2 Packets and frames

Data is sent over networks in discrete chunks called packets (at the IP level) or frames (at the Ethernet level). This is necessary because the network is a shared resource: if one machine sent a continuous stream, no other machine could transmit. Fixed-size packets ensure fairness. The whole idea of the network abstraction is to provide what appears to be a continuous stream of data on top of these fixed-size packets—similar to how a file system provides the illusion of continuous files on top of fixed-size disk blocks.

2.1.3 Internetworking and gateways

A LAN only connects nearby machines. To communicate across LANs, we need internetworking: a virtual network built on top of physical networks. The Internet Protocol (IP) assigns each machine a logical address (the IP address) and uses gateways (routers) to forward packets between networks. This is analogous to virtual memory: just as the kernel builds a virtual address space on top of physical memory, the IP layer builds a virtual global network on top of physical local networks. MAC addresses are the “physical addresses” and IP addresses are the “virtual addresses.”

2.1.4 Reliable communication

IP provides no guarantees: packets can be lost, duplicated, reordered, or corrupted. A reliable transport protocol (like TCP or Plan 9's IL) adds the machinery to provide an ordered, reliable byte stream on top of this unreliable foundation. The key mechanisms are sequence numbers (to detect reordering and loss), acknowledgments (to confirm receipt), and retransmission timers (to resend lost packets). Getting this right is one of the trickiest parts of a network stack, because the protocol must handle arbitrarily delayed, duplicated, and reordered packets.

2.1.5 Network stack

The network stack is the software that implements all these layers. Its main job is multiplexing on the send side (many processes share one network device) and demultiplexing on the receive side (incoming packets must be dispatched to the right process based on addresses and port numbers). It also provides different protocols offering different levels of service: UDP for simple unreliable datagrams, TCP/IL for reliable streams.

2.1.6 Layers

Network stacks are organized in layers, each building on the one below:

```
+--------------------------------+
|  Applications (telnet, hget)  |
+--------------------------------+
|  Transport (TCP, UDP, IL)     |
+--------------------------------+
|  Internet (IP, ICMP)          |
+--------------------------------+
|  Link (Ethernet, ARP)         |
+--------------------------------+
|  Physical (NIC hardware)      |
+--------------------------------+
```

Each layer adds its own header to the data (encapsulation) and strips it on the receiving side. The Plan 9 network stack follows this layered structure closely, with each protocol implemented as a `Proto` module.

2.2 `/net/` files interface

In Plan 9, every kernel service is accessed through files, and networking is no exception. A network connection is a directory of files under `/net/`. To open a TCP connection, you read `/net/tcp/clone` (which allocates a new conversation directory, say `/net/tcp/0/`), write a `connect` command to `/net/tcp/0/ctl`, and then read and write data through `/net/tcp/0/data`. This contrasts with the UNIX socket interface, where connections are created with `socket()`, configured with `ioctl()` or `setsockopt()`, and accessed through a special file descriptor type. In Plan 9, there is no special API—just `open`, `read`, `write`, and `close` on regular files.

2.3 `helloclient.c` and `helloserver.c`

Networking is fundamentally about two programs communicating, so our hello world needs two sides: a server that listens for connections and a client that connects and sends data. Here is a minimal client that sends `hello world` over TCP:

```
/* helloclient.c */
#include <u.h>
#include <libc.h>

void
main(void)
{
    int fd;

    fd = dial("tcp!localhost!9999", nil, nil, nil);
    if(fd < 0)
        sysfatal("dial: %r");
    write(fd, "hello world\n", 12);
    close(fd);
    exits(nil);
}
```

The `dial()` function from `libc` takes a network address in Plan 9's `proto!host!port` syntax and returns a file descriptor for the connection. Under the hood, `dial` opens `/net/tcp/clone` to allocate a conversation, writes a `connect` command to the `ctl` file, and returns the `data` file descriptor. The programmer sees only a simple `open`→`write`→`close` sequence.

Here is the corresponding server:

```
/* helloserver.c */
#include <u.h>
#include <libc.h>

void
main(void)
{
    int acfd, lcfd, dfd;
    char adir[40], ldir[40], buf[128];
    int n;

    acfd = announce("tcp!*!9999", adir);
    if(acfd < 0)
        sysfatal("announce: %r");
    for(;;){
        lcfd = listen(adir, ldir);
        if(lcfd < 0)
            sysfatal("listen: %r");
        dfd = accept(lcfd, ldir);
        if(dfd < 0){
            close(lcfd);
            continue;
        }
        n = read(dfd, buf, sizeof(buf)-1);
        if(n > 0){
            buf[n] = '\0';
            print("received: %s", buf);
```

```
        }
        close(dfd);
        close(lcfd);
    }
}
```

`announce()` binds to port 9999 on all interfaces (`*`), `listen()` waits for an incoming connection, and `accept()` returns a data file descriptor for the new conversation. The server then reads from this file descriptor, prints what it received, and loops. Again, under the hood, these functions are just reading and writing files in `/net/tcp/`—the same interface we explored in the Getting started section.

To test, run the server in one window and the client in another:

```
[window 1]$ helloserver
[window 2]$ helloclient
[window 1] received: hello world
```

2.4 Code organization

The networking code spans three directories. The user-space library `lib_networking/libip/` provides IP address parsing and formatting used by applications. The kernel network stack lives in `kernel/network/ip/`, which contains the protocol implementations (UDP, TCP, IL), the IP layer, ARP, routing, and the `/net` file interface. The Ethernet driver layer is in `kernel/network/386/` (or `arm/` for the Raspberry Pi).

Table 2.1 presents short descriptions of the source files, as well as the main entities (e.g., structures, functions) the file defines, and the corresponding chapter in this document in which the code contained in the file is primarily discussed.

2.5 Software architecture

The networking stack is organized in layers, following the standard Internet model. From bottom to top:

1. **Link layer** (Ethernet): hardware drivers (`devether.c`, `etherigbe.c`) handle frames on the physical wire, with `netif.c` providing a generic interface.

2. **Network layer** (IP): routing (`iproute.c`), ARP (`arp.c`), ICMP (`icmp.c`), and the IP interface management (`ipifc.c`) handle addressing and forwarding of datagrams across networks.

3. **Transport layer** (UDP, TCP, IL): protocol implementations provide either unreliable datagrams (`udp.c`), reliable streams (`tcp.c`), or Plan 9's own reliable datagram protocol (`il.c`).

4. **Application layer**: user programs like `telnet`, `ping`, and `dns` use `dial()`/`announce()` from `libc` to access the stack through `/net` files.

Each layer adds its own header when sending (encapsulation) and strips it when receiving (decapsulation). User programs see a continuous byte stream (TCP) or discrete messages (UDP), but underneath the data travels as Ethernet frames containing IP datagrams, which may be fragmented into multiple packets.

The kernel code uses an object-oriented style despite being written in C. The key abstractions—`Proto`[234b], `Medium`[234b], `Netif`[35f]—are structures with function pointer fields that act as virtual method tables. For instance, each transport protocol registers its own `iput` (input), `kick` (output), `connect`, `announce`, and `close` methods in a `Proto` structure, and the generic `devip` layer dispatches to them without knowing which protocol it is serving.

A few naming conventions appear throughout the code: `f` for the `Fs`[234b] (IP stack instance), `cv` for a `Conv`[234b]ersation (a single connection), `p` for a `Protocol`, and `err` for error strings.

Function	Ch.	File	Entities	LOC
user-space header	3	`include/net/ip.h`	`Ipifc` `Iplifc`	271
kernel header	3	`kernel/.../ip.h`	`Fs`[234b] `Proto`[234b] `Conv`[234b] `Medium`[234b]	1058
netif header	3	`kernel/.../netif.h`	`Netif`[35f] `Netfile`[37a]	197
etherif header	3	`kernel/.../etherif.h`	`Ether`[28e]	73
initialization	4	`ip.c`	`iprouter()X` `Fsproto()`[68a]	812
`/net` file server	5	`devip.c`	`ipopen()`[49a] `ipread()`[51a] `ipwrite()`[51c]	1714
IP interfaces	5	`ipifc.c`	`ipifcbind()`[69] `ipifcadd()`[71c]	1876
netif	7	`netif.c`	`netifread()`[245] `netifwrite()`[246c]	758
Ethernet driver	7	`devether.c`	`etherreset()`[263] `etherwrite()`[259]	623
Intel GbE driver	7	`etherigbe.c`		2193
NE2000 driver	D	`ether2000.c` `ether8390.c`		1129
MII PHY	7	`ethermii.c`		249
Ethernet medium	7	`ethermedium.c`	`ethermediumlink`[366a]	825
other mediums		`nullmedium.c` etc.		287
IP utilities	8	`ipaux.c`	`iptentative()`[378d]	407
ARP	9	`arp.c`	`arpenter()`[338] `arpwrite()`[109b]	739
ICMP		`icmp.c`		550
routing	10	`iproute.c`	`v4lookup()`[112a] `routeadd()X`	986
UDP	11	`udp.c`	`udpiput()`[142] `udpkick()`[140b]	742
IL	12	`il.c`	`iliput()`[158] `ilkick()`[157]	1551
TCP	13	`tcp.c`	`tcpiput()`[441] `tcpoutput()`[449]	3603
kernel dial		`chandial.c`		136
logging		`netlog.c`		299
parsing (user-space)	3	`parseip.c` `parseether.c`	`parseip()`[89] `parseether()`[85a]	203
formatting	A	`eipfmt.c`		118
other libip		`bo.c` `classmask.c` etc.		964
Total				∼22 000

Table 2.1: Chapters and source files of the Plan 9 networking code.

2.5.1 Trace of a network write

2.5.2 Trace of a network read

2.6 Book structure

The rest of the book is organized as follows. Chapter 3 presents the core data structures: IP addresses, the `Fs`[234b]/`Proto`[234b]/`Conv`[234b] hierarchy, media and interfaces, and the kernel-side IP header types. Chapter 4 covers the initialization of the IP stack. Chapter 5 describes the `/net` file-server interface—how user programs interact with the stack by reading and writing files under `/net/tcp/`, `/net/udp/`, etc. Chapter 6 covers the configuration of interfaces and addresses. Chapter 7 presents the Ethernet layer: drivers, the `Netif` abstraction, and the Ethernet medium binding. Chapter 8 covers the IP layer proper. Chapter 9 explains ARP—how IP addresses are resolved to Ethernet MAC addresses on a local network. Chapter 10 covers the routing table and how the stack

decides where to send packets. The transport protocols are presented next: UDP (Chapter 11) for unreliable datagrams, IL (Chapter 12) for Plan 9's own reliable protocol, and TCP (Chapter 13) for the standard reliable stream. The book then covers higher-level topics: applications (Chapter 14), the 9P protocol (Chapter 15), `exportfs` (Chapter 16), DNS (Chapter 17), security (Chapter 18), and advanced topics (Chapter 21). Finally, Chapter 22 concludes. Some appendices present non-functional code: debugging helpers, error management, the NE2000 Ethernet driver, and extra mediums and protocols.

Chapter 3

Core Data Structures

The networking stack has many interconnected data structures. This chapter introduces them all before diving into the code that uses them. The key hierarchy is: `Fs` (the IP stack instance) contains `Protocol`s (UDP, TCP, IL, etc.), each of which manages a set of `Conversation`s (individual connections). On the hardware side, `Ipifc` (IP interface) binds an IP address to a physical `Medium` (like Ethernet), which connects to an `Ether` controller.

3.1 IP addresses

Internally, all IP addresses are stored in the 16-byte IPv6 format (`IPaddrlen` = 16). IPv4 addresses are embedded using the standard IPv4-mapped IPv6 prefix: the first 10 bytes are zero, then two `0xff` bytes, then the 4-byte IPv4 address. The `isv4`, `v4tov6`, and `v6tov4` functions convert between the two representations.

⟨*constant* `IPv4addrlen` 20a⟩≡ (232b 222a)

```
IPv4addrlen= 4,
```

⟨*typedef ipv4* 20b⟩≡ (234b 222c)

```
typedef uchar ipv4[IPv4addrlen];
```

Uses `IPv4addrlen` 20a.

3.1.1 IPv4 vs IPv6

IPv4 uses 4-byte addresses ($2^{32} \approx 4$ billion hosts), which have been exhausted. IPv6 uses 16-byte addresses (2^{128}). Internally, Plan 9 stores all addresses in the 16-byte IPv6 format: IPv4 addresses are embedded with the `::ffff:` prefix (10 zero bytes, two `0xff` bytes, then 4 IPv4 bytes). `v4tov6` and `v6tov4` convert between the two representations; `isv4` checks whether an address is really IPv4 in disguise.

⟨*constant* `IPaddrlen` 20c⟩≡ (232b 222a)

```
IPaddrlen= 16,
```

⟨*typedef ipaddr* 20d⟩≡ (234b 222c)

```
typedef uchar ipaddr[IPaddrlen];
```

Uses `IPaddrlen` 20c.

⟨*global* `v4prefix` 20e⟩≡ (227e)

```
/*
 *  prefix of all v4 addresses
 */
ipaddr v4prefix = {
    // first 12
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0xff, 0xff,
```

```
    // rest are ipv4 numbers
    0, 0, 0, 0
};
```

⟨*constant* IPv4off 21a⟩≡ (232b 222a)

```
IPv4off= 12,
```

⟨*function* isv4 21b⟩≡ (227e)

```
bool
isv4(ipaddr ip)
{
    return memcmp(ip, v4prefix, IPv4off) == 0;
}
```

Uses IPv4off 21a and v4prefix 20e.

⟨*function* v4tov6 21c⟩≡ (227e)

```
/*
 *  the following routines are unrolled with no memset's to speed
 *  up the usual case
 */
void
v4tov6(ipaddr v6, ipv4 v4)
{
    v6[0] = 0; v6[1] = 0; v6[2] = 0; v6[3] = 0;
    v6[4] = 0; v6[5] = 0; v6[6] = 0; v6[7] = 0;
    v6[8] = 0; v6[9] = 0; v6[10] = 0xff; v6[11] = 0xff;

    v6[12] = v4[0];
    v6[13] = v4[1];
    v6[14] = v4[2];
    v6[15] = v4[3];
}
```

⟨*function* v6tov4 21d⟩≡ (227e)

```
errorneg1
v6tov4(ipv4 v4, ipaddr v6)
{
    if(v6[0] == 0 && v6[1] == 0 && v6[2] == 0 && v6[3] == 0
    && v6[4] == 0 && v6[5] == 0 && v6[6] == 0 && v6[7] == 0
    && v6[8] == 0 && v6[9] == 0 && v6[10] == 0xff && v6[11] == 0xff)
    {
        v4[0] = v6[12];
        v4[1] = v6[13];
        v4[2] = v6[14];
        v4[3] = v6[15];
        return OK_0;
    }
    ⟨v6tov4() else if ipv6 address 501a⟩
}
```

⟨*typedef ipv4or6* 21e⟩≡ (234b)

```
typedef uchar* ipv4or6;
```

⟨*typedef ipv4p* 21f⟩≡ (234b)

```
typedef uchar* ipv4p;
```

⟨*enum* ip_version 21g⟩≡ (234b)

```
/* ip versions */
enum ip_version {
  V4=   4,
  V6=   6,
};
```

3.1.2 Network class and network identifier

The original IP address scheme divided the 32-bit address into a network identifier and a host identifier. The split point depends on the "class" (determined by the first 2 bits): class A has 8 bits of network and 24 bits of host, class B has 16/16, and class C has 24/8. The `defmask` function returns the appropriate subnet mask for each class. While classless routing (CIDR) has largely replaced this scheme, the code still supports it as a default.

⟨*macro* CLASS 22a⟩≡ (222c)
```
#define CLASS(p) ((*(uchar*)(p))>>6)
```

3.1.3 Network mask

A network mask separates the network part of an IP address from the host part. `defmask` returns the default mask for the address class: 255.0.0.0 for class A (first byte 0–127), 255.255.0.0 for class B (128–191), and 255.255.255.0 for class C (192–223). `maskip` applies the mask with a byte-by-byte AND: netid = ip & mask. Two machines are on the same local network if their netids match.

⟨*function* defmask 22b⟩≡ (229d)
```
uchar*
defmask(ipaddr ip)
{
    if(isv4(ip))
        return classmask[ip[IPv4off]>>6];
    ⟨defmask() if ipv6 501b⟩
}
```
Uses `IPv4off` 21a, `classmask-5` 22c, and `isv4()` 21b.

⟨*global* classmask 22c⟩≡ (229d)
```
static ipaddr classmask[4] = {
    // class A
    0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,
    0xff,0x00,0x00,0x00, // 255.0.0.0
    // class A
    0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,
    0xff,0x00,0x00,0x00, // 255.0.0.0
    // class B
    0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,
    0xff,0xff,0x00,0x00, // 255.255.0.0
    // class C
    0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,  0xff,0xff,0xff,0xff,
    0xff,0xff,0xff,0x00, // 255.255.255.0
};
```

⟨*function* maskip 22d⟩≡ (229d)
```
void
maskip(ipaddr from, ipaddr mask, ipaddr to)
{
    int i;

    for(i = 0; i < IPaddrlen; i++)
        to[i] = from[i] & mask[i];
}
```
Uses `IPaddrlen` 20c.

3.1.4 Network byte order

Network protocols use big-endian byte order, while the host CPU may use either. The `nhgets`/`hnputs` family of functions (defined in the LIBCORE book [Pad16]) convert between the two. `iplong` is a host-order 32-bit integer used when arithmetic on IP addresses is needed—for instance, checking whether an address falls within a route's address range.

⟨*typedef iplong* 23a⟩≡ (234b 222c)
```
typedef ulong iplong;
```

3.1.5 Other address operations

Since IP addresses are fixed-size byte arrays (not structs), they cannot be assigned with `=` or compared with `==`. The `ipmove` and `ipcmp` macros wrap `memmove` and `memcmp`. `ipcmp` includes a fast-path optimization: it first compares the last byte (which varies most between addresses) before falling through to the full 16-byte comparison.

⟨*global* `IPnoaddr` 23b⟩≡ (227e)
```
ipaddr IPnoaddr;
```

⟨*macro* `ipmove`*(*`(kernel/network/ip/ip.h)`*)* 23c⟩≡ (234b)
```
#define ipmove(x, y) memmove(x, y, IPaddrlen)
```

⟨*macro* `ipcmp`*(*`(kernel/network/ip/ip.h)`*)* 23d⟩≡ (234b)
```
#define ipcmp(x, y) ( (x)[IPaddrlen-1] != (y)[IPaddrlen-1] || memcmp(x, y, IPaddrlen) )
```

3.2 IP header

The `Ip4hdr` structure maps directly to the on-the-wire IPv4 packet header. All multi-byte fields use `uchar` arrays rather than `ushort` or `ulong` to avoid byte-order issues—the network byte order is always big-endian, and using raw bytes with explicit accessor macros (`nhgets`, `hnputs`, etc.) keeps the code portable.

⟨*struct* `Ip4hdr` 23e⟩≡ (234b)
```
/* on the wire packet header */
struct Ip4hdr
{
  uchar vihl;   /* Version and header length */
  uchar tos;    /* Type of service */
  uchar length[2];  /* packet length */
  uchar id[2];    /* ip->identification */
  uchar frag[2];  /* Fragment information */
  uchar ttl;        /* Time to live */

  // enum<protocol_type>
  uchar proto;    /* Protocol */

  uchar cksum[2]; /* Header checksum */

  ipv4 src;   /* IP source */
  ipv4 dst;   /* IP destination */
};
```

3.3 IP Interfaces: `Ipifc`

An `Ipifc` (IP interface) binds one or more IP addresses to a physical device. For example, the command `echo 'bind ether /net/ether0' > /net/ipifc/0/ctl` connects an IP interface to an Ethernet controller. Each interface can have multiple logical addresses (`Iplifc` nodes), which is essential for gateways that bridge different networks.

3.3.1 User side

The user-side `Ipifc` is a read-only view of a network interface, filled by `readipifc` which parses `/net/ipifc`. Each interface has a device name (e.g., "/net/ether0"), an MTU, packet statistics, and a linked list of `Iplifc` entries—the actual IP addresses bound to this interface. Applications use `readipifc` to discover what IP addresses the machine has.

```
⟨struct Ipifc (user) 24a⟩≡                                        (222c)
  /* actual interface */
  struct Ipifc
  {
      /* per ip interface */

      char  dev[64]; // e.g. "/net/ether0"
      int  mtu;
      // list<ref_own<Iplifc> (next = Iplifc.next)
      Iplifc *lifc;

      ⟨Ipifc(user) stat fields 24b⟩
      ⟨Ipifc(user) ipv6 fields 501c⟩

      //Extra
      ⟨Ipifc(user) extra fields 24c⟩
  };
```

```
⟨Ipifc(user) stat fields 24b⟩≡                                     (24a)
  ulong pktin;
  ulong pktout;
  ulong errin;
  ulong errout;
```

```
⟨Ipifc(user) extra fields 24c⟩≡                              (24a) 24d ▷
  Ipifc *next;
```

```
⟨Ipifc(user) extra fields 24d⟩+≡                             (24a) ◁ 24c
  int index;   /* number of interface in ipifc dir */
```

```
⟨struct Iplifc (user) 24e⟩≡                                       (222c)
  /* local address */
  struct Iplifc
  {
      /* per address on the ip interface */
      ipaddr ip;
      ipaddr mask;
      ipaddr net;  /* ip & mask */

      ⟨Iplifc(user) ipv6 fields 516h⟩

      // Extra
      ⟨Iplifc(user) extra fields 25a⟩
  };
```

```
⟨Iplifc(user) extra fields 25a⟩≡                                        (24e)
  // list<ref_own<Iplifc>>, head = Ipifc.lifc
  Iplifc *next;
```

3.3.2 Kernel side

The kernel-side `Ipifc` is richer: it holds the `Medium` pointer, the MAC address, transfer unit bounds, and routing parameters. The `arg` field points to medium-specific private data (for Ethernet, it holds the channel file descriptors for the three packet types: IPv4, ARP, IPv6). The `conv` field links back to the `Conv` in the `ipifc` protocol that owns this interface.

```
⟨struct Ipifc (kernel) 25b⟩≡                                            (234b)
  struct Ipifc
  {
    char  dev[64];  /* device we're attached to */

    Medium  *m;   /* Media pointer */
    uchar mac[MAClen];  /* MAC address */

    int maxtu;    /* Maximum transfer unit */
    int mintu;    /* Minumum tranfer unit */
    int mbps;   /* megabits per second */

    // list<ref_own<Iplifc>>, next = Iplifc.next
    Iplifc  *lifc;    /* logical interfaces on this physical one */

    ⟨Ipifc(kernel) stat fields 25c⟩
    ⟨Ipifc(kernel) routing fields 38f⟩
    ⟨Ipifc(kernel) priv fields 28b⟩

    ⟨Ipifc(kernel) other fields 25d⟩
    ⟨Ipifc(kernel) ipv6 fields 502a⟩

    //Extra
    RWlock;

  };
```

```
⟨Ipifc(kernel) stat fields 25c⟩≡                                        (25b)
  /* message statistics */
  ulong in;
  ulong out;
  ulong inerr;
  ulong outerr;
```

```
⟨Ipifc(kernel) other fields 25d⟩≡                                (25b) 70a ▷
  Conv  *conv;    /* link to its conversation structure */
```

```
⟨struct Iplifc (kernel) 25e⟩≡                                           (234b)
  /* logical interface associated with a physical one */
  struct Iplifc
  {
    ipaddr local;
    ipaddr mask;
    ipaddr net; // local & mask?

    ipaddr remote; // ??

    ⟨Iplifc(kernel) ipv6 fields 516i⟩
```

```
    // Extra
    ⟨Iplifc(kernel) extra fields 26a⟩
  };
```

⟨Iplifc(kernel) *extra fields* 26a⟩≡ (25e) 125d ▷

```
  // list<ref_own<Iplifc>>, head = Ipifc.lifc
  Iplifc  *next;
```

3.4 Physical link

3.4.1 Medium

The `Medium` structure is an interface (in the OOP sense) for physical network types. It provides method pointers for binding to an interface, writing packets (**`bwrite`**), resolving addresses (**`ares`**), and handling multicast. The **`ethermedium`** global is the concrete implementation for Ethernet, with its 14-byte header, 1514-byte MTU, and 6-byte MAC addresses.

⟨*struct Medium (kernel)* 26b⟩≡ (234b)

```
  struct Medium
  {
    char  *name;

    int hsize;     /* medium header size */
    int maclen;    /* mac address length  */

    int mintu;     /* default min mtu */
    int maxtu;     /* default max mtu */

    ⟨Medium(kernel) methods 26c⟩

    ⟨Medium(kernel) other fields 84a⟩

  };
```

⟨Medium(kernel) *methods* 26c⟩≡ (26b)

```
  ⟨Medium(kernel) binding methods 26d⟩
  ⟨Medium(kernel) io methods 26e⟩
  ⟨Medium(kernel) address resolution methods 107a⟩

  ⟨Medium(kernel) router methods 133a⟩
  ⟨Medium(kernel) multicast methods 180g⟩
  ⟨Medium(kernel) ipv6 methods 513a⟩
```

⟨Medium(kernel) *binding methods* 26d⟩≡ (26c)

```
  void  (*bind)(Ipifc*, int, char**);
  void  (*unbind)(Ipifc*);
```

⟨Medium(kernel) *io methods* 26e⟩≡ (26c) 26f ▷

```
  // write packets on the physical network
  void  (*bwrite)(Ipifc *ifc, Block *b, int version, ipv4or6 ip);
```

⟨Medium(kernel) *io methods* 26f⟩+≡ (26c) ◁ 26e

```
  /* process packets written to 'data' */
  void  (*pktin)(Fs *f, Ipifc *ifc, Block *bp);
```

3.4.2 media

⟨*global* `media` 27a⟩≡ (384b)
```
Medium *media[Maxmedia] = { 0 };
```
Uses `Maxmedia-98` 27b.

⟨*constant* `Maxmedia` 27b⟩≡ (374)
```
Maxmedia    = 32,
```

⟨*function* `addipmedium` 27c⟩≡ (384b)
```
/*
 *  link in a new medium
 */
void
addipmedium(Medium *med)
{
    int i;

    for(i = 0; i < nelem(media)-1; i++)
        if(media[i] == nil){
            media[i] = med;
            break;
        }
}
```
Uses `media` 27a.

3.4.3 Ethernet medium

⟨*constant* `Eaddrlen` 27d⟩≡ (240f)
```
Eaddrlen= 6,
```

⟨*typedef eaddr* 27e⟩≡ (241a)
```
typedef uchar eaddr[Eaddrlen];
```
Uses `Eaddrlen` 27d.

⟨*global ethermedium (kernel)* 27f⟩≡ (366c)
```
Medium ethermedium =
{
    .name=      "ether",

    .hsize=     14,
    .mintu=     60,
    .maxtu=     1514,
    .maclen=    6,

    .bind=      etherbind,
    .unbind=    etherunbind,

    .bwrite=    etherbwrite,

    .ares=      arpenter,
    .areg=      sendgarp,

    .addmulti=  etheraddmulti,
    .remmulti=  etherremmulti,

    .pref2addr= etherpref2addr,
};
```
Uses `arpenter()` 338, `etheraddmulti()` 360a, `etherbind()` 77c, `etherpref2addr()` 366b, `etherremmulti()` 360b, `etherunbind()` 357e, and `sendgarp()` 362.

⟨*global* etherbroadcast 28a⟩≡ (366c)
```
static eaddr etherbroadcast = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
```

⟨Ipifc(kernel) *priv fields* 28b⟩≡ (25b)
```
void  *arg;   /* medium specific */
```

3.4.4 Ethernet controllers and etherxx

`etherxx` is a global array of Ethernet controllers, one per physical network card. Each `Ether` structure embeds a `Netif` (for multiplexing) and an `ISAConf` (for hardware I/O port and IRQ information). The `ctlr` field points to the driver-specific control block (e.g., `Dp8390` for NE2000). The `transmit` method is called to send a packet; `interrupt` is the hardware interrupt handler.

⟨*global etherxx (kernel)* 28c⟩≡ (264b)
```
static Ether *etherxx[MaxEther];
```
Uses MaxEther 28d.

⟨*constant* MaxEther 28d⟩≡ (255a)
```
MaxEther  = 48, //bcm: 4
```

⟨*struct Ether (kernel)* 28e⟩≡ (255d)
```
struct Ether {

  eaddr ea;

  Queue*  oq;

  ISAConf;      /* hardware info */
  Netif;

  ⟨Ether methods 28i⟩

  ⟨Ether priv fields 28g⟩
  ⟨Ether other fields 28h⟩

  // Extra
  ⟨Ether extra fields 28f⟩

  //bcm: only, but seems dead
  //RWlock;
  //int minmtu;
  //int maxmtu;
  //void* address;
  //int irq;

};
```

⟨Ether *extra fields* 28f⟩≡ (28e)
```
int ctlrno;
```

⟨Ether *priv fields* 28g⟩≡ (28e)
```
void  *ctlr;
```

⟨Ether *other fields* 28h⟩≡ (28e)
```
int tbdf;     /* type+busno+devno+funcno */ //pc: only?
```

⟨Ether *methods* 28i⟩≡ (28e)
```
⟨ether mounting methods 29a⟩
⟨ether io methods 29b⟩

⟨ether other methods 29c⟩
```

⟨ether *mounting methods* 29a⟩≡ (28i)
```
  void  (*attach)(Ether*);  /* filled in by reset routine */
  void  (*detach)(Ether*);
```

⟨ether *io methods* 29b⟩≡ (28i)
```
  void  (*transmit)(Ether*);
  void  (*interrupt)(Ureg*, void*);
```

⟨ether *other methods* 29c⟩≡ (28i)
```
  long  (*ifstat)(Ether*, void*, long, ulong);
  long  (*ctl)(Ether*, void*, long); /* custom ctl messages */
  void  (*power)(Ether*, int);  /* power on/off */
  void  (*shutdown)(Ether*);  /* shutdown hardware before reboot */
```

3.5 /net filesystem

3.5.1 Fs and ipfs

`Fs` is the root of the entire IP stack. It holds a pointer to the `IP` fragment reassembly state, an array of registered protocols (`p`), the ARP cache, the routing tables, and the network database. The global `ipfs` array allows multiple independent IP stacks (indexed by device number), though in practice only one is used.

⟨*struct Fs (kernel)* 29d⟩≡ (234b)
```
  /*
   *  one per IP protocol stack
   */
  struct Fs
  {
    IP  *ip;

    // array<option<ref_own<Proto>>>, size is Fs.np
    Proto*  p[Maxproto+1];    /* list of supported protocols */
    int np;

    ⟨Fs(kernel) arp fields 106a⟩
    ⟨Fs(kernel) routing fields 30b⟩
    ⟨Fs(kernel) ndb fields 181f⟩
    ⟨Fs(kernel) logging fields 189b⟩

    ⟨Fs(kernel) other fields 32c⟩
    ⟨Fs(kernel) ipv6 fields 501e⟩

    // Extra
    RWlock;
    ⟨Fs(kernel) extra fields 29h⟩
  };
```

⟨*constant* `Maxproto` 29e⟩≡ (232b)
```
  Maxproto= 20,
```

⟨*global* `ipfs` 29f⟩≡ (331c)
```
  Fs  *ipfs[Nfs]; /* attached fs's */
```
Uses `Nfs-244` 29g.

⟨*constant* `Nfs` 29g⟩≡ (35a)
```
  Nfs=        128,
```

⟨`Fs(kernel)` *extra fields* 29h⟩≡ (29d)
```
  int dev; // idx in ipfs
```

⟨*global* fslock 30a⟩≡ (331c)
```
QLock   fslock;
```

⟨Fs(kernel) *routing fields* 30b⟩≡ (29d) 39a ▷
```
Proto*  ipifc;      /* kludge for ipifcremroute & ipifcaddroute */
```

3.5.2 IP fragments and IP statistics

The IP structure manages IP-level state: fragment reassembly and MIB-II statistics. When a packet exceeds the MTU of a network link, IP splits it into fragments that are independently routed and reassembled at the destination. The `Fragment4` pool is a pre-allocated free list of fragment tracking structures, linked through `next` pointers.

⟨*struct IP (kernel)* 30c⟩≡ (234b)
```
/* an instance of IP */
struct IP
{
  // list<ref_own<Fragment>>, next = Fragment4.next
  Fragment4*  flisthead4;
  // list<ref_own<Fragment>>, next = Fragment4.next
  Fragment4*  fragfree4;

  Ref   id4;

  ⟨IP(kernel) stat fields 30f⟩
  ⟨IP(kernel) routing fields 105a⟩

  ⟨IP(kernel) ipv6 fields 501f⟩

  // Extra
  QLock   fraglock4;
};
```

⟨*struct* Fragment4 30d⟩≡ (234b)
```
struct Fragment4
{
  // list<ref_own<Block>> ? next = Block.???
  Block*  blist;

  iplong   src;
  iplong   dst;

  ushort  id;
  ulong   age;

  // Extra
  ⟨Fragment4 extra fields 30e⟩
};
```

⟨Fragment4 *extra fields* 30e⟩≡ (30d)
```
Fragment4*  next;
```

⟨IP(kernel) *stat fields* 30f⟩≡ (30c)
```
// map<enum<mib_two_counters>, uvlong>
uvlong    stats[Nipstats];
```

⟨*enum* _anon_ (kernel/network/ip/ip.h)3 31a⟩≡ (234b)

```
/* MIB II counters */
enum mib_two_counters
{
  Forwarding,
  DefaultTTL,

  InReceives,

  InHdrErrors,
  InAddrErrors,
  ForwDatagrams,
  InUnknownProtos,

  // In stats
  InDiscards,
  InDelivers,

  // Out stats
  OutRequests,
  OutDiscards,
  OutNoRoutes,

  ReasmTimeout,
  ReasmReqds,
  ReasmOKs,
  ReasmFails,

  // Out fragments
  FragOKs,
  FragFails,
  FragCreates,

  Nipstats,
};
```

⟨*global* statnames 31b⟩≡ (239a)

```
static char *statnames[] =
{
[Forwarding]    "Forwarding",
[DefaultTTL]    "DefaultTTL",
[InReceives]    "InReceives",
[InHdrErrors]   "InHdrErrors",
[InAddrErrors]  "InAddrErrors",
[ForwDatagrams] "ForwDatagrams",
[InUnknownProtos]    "InUnknownProtos",

[InDiscards]    "InDiscards",
[InDelivers]    "InDelivers",

[OutRequests]   "OutRequests",
[OutDiscards]   "OutDiscards",
[OutNoRoutes]   "OutNoRoutes",

[ReasmTimeout]  "ReasmTimeout",
[ReasmReqds]    "ReasmReqds",
[ReasmOKs]  "ReasmOKs",
[ReasmFails]    "ReasmFails",

[FragOKs]   "FragOKs",
```

```
[FragFails] "FragFails",
[FragCreates]   "FragCreates",
};
```

3.5.3 Protocols

A `Proto` is a protocol implementation registered with the IP stack. It provides methods for creating and closing connections (`create`, `close`), establishing communication (`connect`, `announce`), and receiving packets (`rcv`). Each protocol manages an array of conversations and has a protocol number (`ipproto`) used to dispatch incoming packets—the `t2p` table in `Fs` maps the protocol field in IP headers to the correct `Proto`.

⟨*struct Proto (kernel)* 32a⟩≡ (234b)

```
/*
 *  one per multiplexed protocol
 */
struct Proto
{
  char*   name;   /* protocol name */ // e.g. "udp", "tcp", "ipfc", etc

  ⟨Proto(kernel) methods 32f⟩

  // growing_array<option<ref_own<Proto>>>, size = Proto.nc
  Conv    **conv;   /* array of conversations */
  int   nc;   /* number of conversations */
  int   ac; // number of opened conversations

  ⟨Proto(kernel) priv fields 32e⟩
  ⟨Proto(kernel) other fields 32d⟩

  // Extra
  QLock;
  ⟨Proto(kernel) extra fields 32b⟩
};
```

⟨`Proto(kernel)` *extra fields* 32b⟩≡ (32a)

```
// ref<Fs>, reverse of Fs.p[this.x]
Fs    *f;   /* file system this proto is part of */
// index in Fs.p[]
int   x;    /* protocol index */
```

⟨`Fs(kernel)` *other fields* 32c⟩≡ (29d) 125c ▷

```
// map<enum<protocol_type>, ref<Proto>>
Proto*  t2p[256];   /* vector of all protocols */
```

⟨`Proto(kernel)` *other fields* 32d⟩≡ (32a) 58e ▷

```
// enum<protocol_type>
int   ipproto;  /* ip protocol type */
```

⟨`Proto(kernel)` *priv fields* 32e⟩≡ (32a) 34g ▷

```
void    *priv;
```

⟨`Proto(kernel)` *methods* 32f⟩≡ (32a)

```
⟨Proto(kernel) protocol methods 32g⟩
⟨Proto(kernel) conversation ctl methods 33b⟩
⟨Proto(kernel) conversation inspection methods 64f⟩
⟨Proto(kernel) conversation methods 33a⟩
```

⟨`Proto(kernel)` *protocol methods* 32g⟩≡ (32f) 54c ▷

```
void    (*create)(Conv*);
void    (*close)(Conv*);
```

⟨Proto(kernel) *conversation methods* 33a⟩≡ (32f) 54a▷
```
void    (*rcv)(Proto*, Ipifc*, Block*);
```

⟨Proto(kernel) *conversation ctl methods* 33b⟩≡ (32f) 33c▷
```
char*   (*connect)(Conv*, char**, int);
char*   (*announce)(Conv*, char**, int);
```

⟨Proto(kernel) *conversation ctl methods* 33c⟩+≡ (32f) ◁33b 33d▷
```
char*   (*ctl)(Conv*, char**, int);
```

⟨Proto(kernel) *conversation ctl methods* 33d⟩+≡ (32f) ◁33c
```
char*   (*bind)(Conv*, char**, int);
```

3.5.4 Conversations

A `Conv` (conversation) represents one network connection, identified by the 4-tuple of local IP, local port, remote IP, and remote port. It is the central structure of the stack—the equivalent of a UNIX socket. Each conversation has a read queue (`rq`) for incoming data and a write queue (`wq`) for outgoing data, plus a state machine tracking whether the connection is idle, connecting, or established. Conversations are exposed as directories under `/net/<proto>/<n>/` with `data`, `ctl`, `local`, `remote`, and `status` files.

⟨*struct Conv (kernel)* 33e⟩≡ (234b)
```
/*
 *  one per conversation directory
 */
struct Conv
{

  ipaddr laddr; /* local IP address */
  ipaddr raddr; /* remote IP address */

  ushort  lport;      /* local port number */
  ushort  rport;      /* remote port number */

  char  *owner;     /* protections */
  int perm;

  // enum<conversation_state>
  int state;

  ⟨Conv(kernel) queue fields 34d⟩
  ⟨Conv(kernel) listen fields 63a⟩
  ⟨Conv(kernel) routing fields 37b⟩

  ⟨Conv(kernel) synchronisation fields 34c⟩
  ⟨Conv(kernel) priv fields 34f⟩
  ⟨Conv(kernel) snoop fields 192b⟩
  ⟨Conv(kernel) error fields 34e⟩

  ⟨Conv(kernel) udp fields 147d⟩
  ⟨Conv(kernel) multicast fields 181a⟩

  ⟨Conv(kernel) other fields 34h⟩
  ⟨Conv(kernel) ipv6 fields 516e⟩

  // Extra
  QLock;
  ⟨Conv(kernel) extra fields 34a⟩
};
```

⟨Conv(kernel) *extra fields* 34a⟩≡ (33e)
```
// ref<Proto> reverse of Proto.conv[this.x]
Proto*  p;
// index in Proto.conv[]
int x;      /* conversation index */
```

⟨*enum* conversation_state 34b⟩≡ (234b)
```
enum conversation_state
{
  Idle=   0,

  Announcing= 1,
  Announced=  2,

  Connecting= 3,
  Connected=  4,
};
```

⟨Conv(kernel) *synchronisation fields* 34c⟩≡ (33e)
```
Rendez  cr;
```

⟨Conv(kernel) *queue fields* 34d⟩≡ (33e) 62a ▷
```
Queue*  rq;     /* queued data waiting to be read */
Queue*  wq;     /* queued data waiting to be written */
```

⟨Conv(kernel) *error fields* 34e⟩≡ (33e)
```
char  cerr[ERRMAX];
```

⟨Conv(kernel) *priv fields* 34f⟩≡ (33e)
```
void* ptcl;     /* protocol specific stuff */
```

⟨Proto(kernel) *priv fields* 34g⟩+≡ (32a) ◁ 32e
```
int   ptclsize; /* size of per protocol ctl block */
```

⟨Conv(kernel) *other fields* 34h⟩≡ (33e) 58d ▷
```
int inuse;      /* opens of listen/data/ctl */
```

3.5.5 Chan Qid, `PROTO()`, `CONV()`, `QID()`

The `/net` filesystem encodes three coordinates into a single Qid path: 8 bits for the protocol index, 12 bits for the conversation number, and 5 bits for the file type (ctl, data, etc.). The `QID` macro packs these three values; `PROTO`, `CONV`, and `TYPE` extract them. This scheme allows the device's `walk`/`open`/`read`/`write` functions to identify exactly which protocol, conversation, and file a channel refers to with a single integer comparison.

⟨*enum* qid*(*kernel/network/ip/devip.c*)* 34i⟩≡ (331c)
```
enum
{
    Qtopdir=    1,      /* top level directory */

    Qtopbase,
    Qarp=       Qtopbase,
    Qiproute,
    ⟨Qid toplevel extra cases 130a⟩

    Qprotodir,          /* directory for a protocol */
    Qprotobase,
    Qclone=     Qprotobase,
    ⟨Qid protocol extra cases 63c⟩
```

```
    Qconvdir,           /* directory for a conversation */
    Qconvbase,
    Qctl=       Qconvbase,
    Qdata,
    ⟨Qid conversation extra cases 62b⟩
    ⟨Qid conversation extra cases, last entry 192a⟩
};
```

Uses Qconvbase-227 34i, Qprotobase-223 34i, and Qtopbase-216 34i.

⟨*enum* misc*(*(kernel/network/ip/devip.c)*)* 35a⟩≡ (331c)

```
enum
{
    Logtype=    5,
    Masktype=   (1<<Logtype)-1,

    Logconv=    12,
    Maskconv=   (1<<Logconv)-1,
    Shiftconv=  Logtype,

    Logproto=   8,
    Maskproto=  (1<<Logproto)-1,
    Shiftproto= Logtype + Logconv,

    ⟨constant Nfs 29g⟩
};
```

Uses Logconv-238 35a, Logproto-241 35a, and Logtype-236 35a.

⟨*macro* QID 35b⟩≡ (331c)

```
#define QID(p, cv, y)    ( ((p)<<(Shiftproto)) | ((cv)<<Shiftconv) | (y) )
```

⟨*macro* PROTO 35c⟩≡ (331c)

```
#define PROTO(x)    ( (((ulong)(x).path) >> Shiftproto) & Maskproto )
```

⟨*macro* CONV 35d⟩≡ (331c)

```
#define CONV(x)     ( (((ulong)(x).path) >> Shiftconv) & Maskconv )
```

⟨*macro* TYPE 35e⟩≡ (331c)

```
#define TYPE(x)     ( ((ulong)(x).path) & Masktype )
```

3.6 /net/ether filesystem

3.6.1 Netif

Netif is the generic network interface structure, embedded (via C structure embedding) in every Ether controller. It provides protocol-independent multiplexing: multiple processes can open the same Ethernet device and filter packets by type. Each open gets a Netfile with its own input queue. Netif also tracks link statistics (packets in/out, CRC errors, overflows) and supports promiscuous mode for packet sniffing.

⟨*struct Netif (kernel)* 35f⟩≡ (241a)

```
/*
 *  a network interface
 */
struct Netif
{
  /* multiplexing */
  char  name[KNAMELEN];   /* for top level directory */
```

```
    // growing_array?<option<ref_own<Netfile>>>, size = nfile?
    Netfile **f;
    int nfile;      /* max number of Netfiles */

    /* about net */
    int alen;     /* address length */
    int link;     /* link status */

    int limit;      /* flow control */

    int minmtu;
    int maxmtu;
    int mtu;
    int mbps;     /* megabits per sec */

    uchar addr[Nmaxaddr];
    uchar bcast[Nmaxaddr];

    int prom;     /* number of promiscuous opens */
    int scan;     /* number of base station scanners */
    int all;      /* number of -1 multiplexors */

    ⟨Netif(kernel) stat fields 36b⟩

    /* routines for touching the hardware */
    void  *arg;

    ⟨Netif(kernel) methods 36a⟩

    // Extra
    QLock;

  };
```

Uses Nmaxaddr 240a.

⟨Netif(kernel) *methods* 36a⟩≡ (35f)

```
  void  (*promiscuous)(void*, int);
  int (*hwmtu)(void*, int); /* get/set mtu */
  void  (*scanbs)(void*, uint); /* scan for base stations */

  ⟨Netif(kernel) multicast fields 181d⟩
  ⟨Netif(kernel) multicast methods 181e⟩
```

⟨Netif(kernel) *stat fields* 36b⟩≡ (35f)

```
  /* statistics */
  int misses;
  uvlong  inpackets;
  uvlong  outpackets;
  int crcs;   /* input crc errors */
  int oerrs;    /* output errors */
  int frames;   /* framing errors */
  int overflows;  /* packet overflows */
  int buffs;    /* buffering errors */
  int soverflows; /* software overflow */
```

3.6.2 Netfile

Each open of `/net/etherX/N` creates a `Netfile` that acts as a filter: the `type` field selects which Ethernet frame types this file receives (e.g., only IP packets, only ARP packets). The `prom` flag enables promiscuous mode for

this file, allowing packet capture. Incoming frames are demultiplexed by `etheriq` and copied into the `in` queue of each matching `Netfile`.

⟨*struct* `Netfile` 37a⟩≡ (241a)
```
/*
 *  one per multiplexed connection
 */
struct Netfile
{
  int inuse;
  ulong mode;
  char  owner[KNAMELEN];

  Queue *in;      /* input buffer */

  int type;     /* multiplexor type */
  int prom;     /* promiscuous mode */
  int scan;     /* base station scanning interval */
  int bridge;     /* bridge mode */
  int headersonly;    /* headers only - no data */

  uchar maddr[8];   /* bitmask of multicast addresses requested */
  int nmaddr;     /* number of multicast addresses */

  // Extra
  QLock;
};
```

3.7 Routes

Routes connect the IP layer to the physical layer: given a destination IP address, `v4lookup` finds the `Route` that tells which gateway to forward through and which `Ipifc` to send on. Each `Conv` caches its last route in `Conv.r` with a generation counter (`rgen`) to detect when the routing table has changed. The routing table is a binary search forest keyed by IP address ranges, with 1024 root buckets for fast lookup.

3.7.1 Single Route

⟨`Conv(kernel)` *routing fields* 37b⟩≡ (33e) 37c ▷
```
Route *r;     /* last route used */
```

⟨`Conv(kernel)` *routing fields* 37c⟩+≡ (33e) ◁ 37b
```
ulong rgen;     /* routetable generation for *r */
```

⟨*global* `v4routegeneration` 37d⟩≡ (369c)
```
static ulong v4routegeneration;
```

⟨*struct Route (kernel)* 37e⟩≡ (234b)
```
struct Route
{
  RouteTree;

  union {
    V4route v4;
    ⟨Route ipv6 route union case 516d⟩
  };

};
```

⟨*struct* V4route 38a⟩≡ (234b)
```
struct V4route
{
  iplong address;
  iplong endaddress;

  ipv4 gate;
};
```

⟨*struct RouteTree (kernel)* 38b⟩≡ (234b)
```
struct  RouteTree
{
  // bitset<enum<route_type> >
  uchar type;

  Ipifc *ifc; // !!! the missing link!

  ⟨Routetree other fields 38c⟩

  // Extra
  Route*  right;
  Route*  left;
  Route*  mid;

};
```

⟨Routetree *other fields* 38c⟩≡ (38b) 38d ▷
```
char  tag[4];
```

⟨Routetree *other fields* 38d⟩+≡ (38b) ◁ 38c 70b ▷
```
int ref;
```

⟨*enum* _anon_ (kernel/network/ip/ip.h)6 38e⟩≡ (234b)
```
enum route_type
{
  /* type bits */
  Rv4=    (1<<0),   /* this is a version 4 route */

  Rifc=   (1<<1),   /* this route is a directly connected interface */
  Rptpt=    (1<<2),   /* this route is a pt to pt interface */

  Runi=   (1<<3),   /* a unicast self address */

  Rbcast=   (1<<4),   /* a broadcast self address */
  Rmulti=   (1<<5),   /* a multicast self address */

  Rproxy=   (1<<6),   /* this route should be proxied */
};
```

⟨Ipifc(kernel) *routing fields* 38f⟩≡ (25b)
```
Routerparams rp;  /* router parameters as in RFC 2461, pp.40|43.
        used only if node is router */
```

3.7.2 Routing forest

The routing table is a forest of $2^{10} = 1024$ ternary search trees. The destination IP address is hashed (via `V4H`) to select a root, then the tree is walked using address-range comparisons: left for ranges entirely below,

right for ranges entirely above, and mid for overlapping ranges (more specific routes). This structure supports longest-prefix matching efficiently.

⟨Fs(kernel) *routing fields* 39a⟩+≡ (29d) ◁30b 118c▷
```
// hash<ip, ref<Route>> where hash function is V4H
Route *v4root[1<<Lroot];  /* v4 routing forest */
```

⟨*constant* Lroot 39b⟩≡ (232b)
```
/* 2^Lroot trees in the root table */
Lroot=    10,
```

3.8 Blocks

Network data travels through the kernel as `Block`s (see the Kernel book [Pad14]). The `BLKIP` macro casts a block's read pointer to an `Ip4hdr` for direct access to IP header fields, and `BLKIPVER` extracts the IP version nibble to distinguish IPv4 from IPv6 packets.

⟨*macro* BLKIPVER 39c⟩≡ (239a)
```
#define BLKIPVER(xp)    (((Ip4hdr*)((xp)->rp))->vihl&0xF0)
```

⟨*macro* BLKIP 39d⟩≡ (239a)
```
#define BLKIP(xp)   ((Ip4hdr*)((xp)->rp))
```

Chapter 4

Initialization

Initializing the network stack is a two-stage process. First, the kernel mounts the IP device (`bind #I /net`) which creates the `Fs` structure and initializes all protocols, ARP, and fragment tracking. Then, the Ethernet device is mounted (`bind #l /net`) to provide the physical transport. On the user side, `ipconfig` configures the IP address and routes, typically via DHCP.

4.1 Kernel side

4.1.1 Mounting the ip device

Mounting the IP device (`bind #I /net`) triggers `ipattach`, which calls `ipgetfs` to create or reuse an `Fs` structure. `ipgetfs` is the real initialization: it allocates the `Fs`, creates the `IP` fragment tracking structure, initializes the ARP cache, and calls each protocol's init function (UDP, TCP, IL, ICMP, `ipifc`, etc.) to register them. After this, `/net` is populated with protocol directories (`/net/tcp/`, `/net/udp/`, `/net/ipifc/`, etc.) and special files (`/net/arp`, `/net/iproute`).

⟨*function* `ipattach` 40a⟩≡ (331c)

```
static Chan*
ipattach(char* spec)
{
    Chan *c;
    int dev;

    dev = atoi(spec);
    if(dev >= Nfs)
        error("bad specification");

    // initialize ip stack
    ipgetfs(dev);

    c = devattach('I', spec);
    mkqid(&c->qid, QID(0, 0, Qtopdir), 0, QTDIR);
    c->dev = dev;
    c->aux = newipaux(up->user, "none");

    return c;
}
```

Uses `Nfs-244` 29g, `QID-248` 35b, `Qtopdir-215` 34i, `ipgetfs()` 40b, and `newipaux()` 41c.

⟨*function* `ipgetfs` 40b⟩≡ (331c)

```
static Fs*
ipgetfs(int dev)
{
```

```
    extern void (*ipprotoinit[])(Fs*);
    Fs *f;
    int i;

    if(dev >= Nfs)
        return nil;

    qlock(&fslock);
    if(ipfs[dev] == nil){
        f = smalloc(sizeof(Fs));

        ip_init(f);
        arpinit(f);
        netloginit(f);
        for(i = 0; ipprotoinit[i]; i++)
            ipprotoinit[i](f);

        f->dev = dev;
        ipfs[dev] = f;
    }
    qunlock(&fslock);

    return ipfs[dev];
}
```

Uses `Nfs-244` 29g, `arpinit()` 107b, `fslock` 30a, `ip_init()` 42a, `ipfs` 29f, and `netloginit()` 189c.

⟨*struct* `IPaux` 41a⟩≡ (234b)

```
/*
 *  Hanging off every ip channel's ->aux is the following structure.
 *  It maintains the state used by devip and iproute.
 */
struct IPaux
{
  char  *owner;   /* the user that did the attach */
  char  tag[4];
};
```

⟨*macro* `ATTACHER` 41b⟩≡ (331c)

```
#define ATTACHER(c) (((IPaux*)((c)->aux))->owner)
```

⟨*function* `newipaux` 41c⟩≡ (331c)

```
IPaux*
newipaux(char *owner, char *tag)
{
    IPaux *a;
    int n;

    a = smalloc(sizeof(IPaux));

    kstrdup(&a->owner, owner);
    memset(a->tag, ' ', sizeof(a->tag));
    n = strlen(tag);
    if(n > sizeof(a->tag))
        n = sizeof(a->tag);
    memmove(a->tag, tag, n);

    return a;
}
```

ip_init()

⟨*function* ip_init 42a⟩≡ (239a)

```
void
ip_init(Fs *f)
{
    IP *ip;

    ip = smalloc(sizeof(IP));
    initfrag(ip, 100);
    f->ip = ip;

    ⟨ip_init() ipv6 init 505e⟩
}
```

Uses initfrag() 42b.

⟨*function* initfrag 42b⟩≡ (239a)

```
void
initfrag(IP *ip, int size)
{
    Fragment4 *fq4, *eq4;
    ⟨initfrag() locals 505f⟩

    ip->fragfree4 = (Fragment4*)malloc(sizeof(Fragment4) * size);
    if(ip->fragfree4 == nil)
        panic("initfrag");

    eq4 = &ip->fragfree4[size];
    for(fq4 = ip->fragfree4; fq4 < eq4; fq4++)
        fq4->next = fq4+1;
    ip->fragfree4[size-1].next = nil;

    ⟨initfrag() ipv6 init fragfree6 505g⟩
}
```

arpinit()

netloginit()

ipprotoinit()

4.1.2 Mounting the ethernet device

Mounting the Ethernet device (`bind #l0 /net`) triggers `etherattach`, which looks up controller 0 in the `etherxx` array and calls its `attach` method. This populates `/net/ether0/` with files for reading/writing Ethernet frames, the MAC address (`/net/ether0/addr`), and statistics. The MAC address can be read with `cat /net/ether0/addr`.

⟨*function* etherattach 42c⟩≡ (264b)

```
Chan*
etherattach(char* spec)
{
    ulong ctlrno;
    char *p;
    Chan *chan;

    ctlrno = 0;
    if(spec && *spec){
        ctlrno = strtoul(spec, &p, 0);
```

```
        if((ctlrno == 0 && p == spec) || *p || (ctlrno >= MaxEther))
            error(Ebadarg);
    }
    if(etherxx[ctlrno] == nil)
        error(Enodev);

    chan = devattach('l', spec);
    if(waserror()){
        chanfree(chan);
        nexterror();
    }
    chan->dev = ctlrno;
    if(etherxx[ctlrno]->attach)
        // Ethernet controller dispatch
        etherxx[ctlrno]->attach(etherxx[ctlrno]);

    poperror();
    return chan;
}
```

4.2 User side

4.2.1 Connect manually

A user program connects manually by opening `/net/tcp/clone` (which returns a conversation number), then writing `connect 10.0.0.1!80` to `/net/tcp/N/ctl`, and reading/writing `/net/tcp/N/data`. The `dial` library function wraps this sequence, optionally consulting the connection server for name resolution.

4.2.2 `dial()`

Chapter 5

User/Kernel Bridge

In Plan 9, all network operations go through the file system. The `ipdevtab` is a kernel device that serves the `/net` file tree. When a user program opens `/net/tcp/clone`, the kernel calls `ipopen` which allocates a new conversation. Writing `connect 10.0.0.1!80` to the `ctl` file triggers `ipwrite`, which dispatches to the protocol's `connect` method. Reading from `data` dequeues packets from `rq`. This chapter traces how file operations are dispatched through the `/net` hierarchy to the protocol implementations.

5.1 IP device

⟨*global* ipdevtab 44a⟩≡ (331c)

```
Dev ipdevtab = {
    .dc       =    'I',
    .name     =    "ip",

    .attach   =    ipattach,
    .walk     =    ipwalk,
    .open     =    ipopen,
    .close    =    ipclose,
    .read     =    ipread,
    .write    =    ipwrite,
    .stat     =    ipstat,
    .wstat    =    ipwstat,

    .create   =    ipcreate,
    .remove   =    ipremove,
    .bread    =    ipbread,
    .bwrite   =    ipbwrite,
    .reset    =    ipreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
};
```

Uses `ipattach()` 40a, `ipbread()` 329b, `ipbwrite()` 330b, `ipclose()` 50a, `ipcreate()` 44b, `ipopen()` 49a, `ipread()` 51a, `ipremove()` 44c, `ipreset()` 327, `ipstat()` 328a, `ipwalk()` 45a, `ipwrite()` 51c, and `ipwstat()` 328c.

⟨*function* ipcreate 44b⟩≡ (331c)

```
static void
ipcreate(Chan*, char*, int, ulong)
{
    error(Eperm);
}
```

⟨*function* ipremove 44c⟩≡ (331c)

```
static void
```

```
ipremove(Chan*)
{
    error(Eperm);
}
```

5.1.1 /net hierarchy and ipwalk()

`ipgen` generates the directory listing for each level of the `/net` hierarchy. The Qid encoding packs the protocol index, conversation index, and file type into a single 64-bit identifier, allowing `ipgen` to determine from any Qid what level of the tree it's at and what entries to show.

⟨*function* ipwalk 45a⟩≡ (331c)

```
static Walkqid*
ipwalk(Chan* c, Chan *nc, char **name, int nname)
{
    IPaux *a = c->aux;
    Walkqid* w;

    w = devwalk(c, nc, name, nname, nil, 0, ipgen);

    if(w != nil && w->clone != nil)
        w->clone->aux = newipaux(a->owner, a->tag);

    return w;
}
```

Uses `ipgen()` 45b and `newipaux()` 41c.

⟨*function* ipgen 45b⟩≡ (331c)

```
static int
ipgen(Chan *c, char*, Dirtab*, int, int s, Dir *dp)
{
    Fs *f;
    Qid q;
    Conv *cv;

    f = ipfs[c->dev];
    switch(TYPE(c->qid)) {
    ⟨ipgen() switch TYPE qid cases 45c⟩
    }
    return -1;
}
```

Uses `TYPE-245` 35e and `ipfs` 29f.

/net/

⟨`ipgen()` *switch TYPE qid cases* 45c⟩≡ (45b) 46b▷

```
case Qtopdir:
    if(s == DEVDOTDOT){
        mkqid(&q, QID(0, 0, Qtopdir), 0, QTDIR);
        snprint(up->genbuf, sizeof up->genbuf, "#I%lud", c->dev);
        devdir(c, q, up->genbuf, 0, network, 0555, dp);
        return 1;
    }
    if(s < f->np) {
        if(f->p[s]->connect == nil)
            return 0;   /* protocol with no user interface */
        mkqid(&q, QID(s, 0, Qprotodir), 0, QTDIR);
        devdir(c, q, f->p[s]->name, 0, network, 0555, dp);
```

```
        return 1;
    }
    s -= f->np;
    return ip1gen(c, Qtopbase+s, dp);
```
Uses QID-248 35b, Qprotodir-222 34i, Qtopbase-216 34i, Qtopdir-215 34i, ip1gen() 46c, and network-249 46a.

⟨*global* network 46a⟩≡ (331c)
```
  static char network[] = "network";
```
Uses network-249 46a.

/net/xxx

⟨ipgen() *switch TYPE qid cases* 46b⟩+≡ (45b) ◁45c 46d▷
```
  case Qarp:
  case Qiproute:
  case Qipselftab:
  case Qndb:
  case Qlog:
      return ip1gen(c, TYPE(c->qid), dp);
```
Uses Qarp-217 34i, Qiproute-218 34i, Qipselftab-219 130a, Qlog-221 189d, Qndb-220 181g, TYPE-245 35e, and ip1gen() 46c.

⟨*function* ip1gen 46c⟩≡ (331c)
```
  static int
  ip1gen(Chan *c, int i, Dir *dp)
  {
      Fs *f;
      Qid q;
      int prot;
      int len = 0;
      char *p;
      ⟨ip1gen() locals 181i⟩

      f = ipfs[c->dev];
      prot = 0666;
      mkqid(&q, QID(0, 0, i), 0, QTFILE);
      switch(i) {
      ⟨ip1gen() switch TYPE qid cases 108a⟩
      default:
          return -1;
      }
      devdir(c, q, p, len, network, prot, dp);
      ⟨ipgen() if Qndb, adjust mtime 181j⟩
      return 1;
  }
```
Uses QID-248 35b, ipfs 29f, and network-249 46a.

/net/proto/

⟨ipgen() *switch TYPE qid cases* 46d⟩+≡ (45b) ◁46b 47a▷
```
  case Qprotodir:
      if(s == DEVDOTDOT){
          mkqid(&q, QID(0, 0, Qtopdir), 0, QTDIR);
          snprint(up->genbuf, sizeof up->genbuf, "#I%lud", c->dev);
          devdir(c, q, up->genbuf, 0, network, 0555, dp);
          return 1;
      }
      if(s < f->p[PROTO(c->qid)]->ac) {
          cv = f->p[PROTO(c->qid)]->conv[s];
```

```
        snprint(up->genbuf, sizeof up->genbuf, "%d", s);
        mkqid(&q, QID(PROTO(c->qid), s, Qconvdir), 0, QTDIR);
        devdir(c, q, up->genbuf, 0, cv->owner, 0555, dp);
        return 1;
    }
    s -= f->p[PROTO(c->qid)]->ac;
    return ip2gen(c, s+Qprotobase, dp);
```

Uses PROTO-247 35c, QID-248 35b, Qconvdir-226 34i, Qprotobase-223 34i, Qprotodir-222 34i, Qtopdir-215 34i, ip2gen() 47b, and network-249 46a.

/net/proto/xxx

⟨ipgen() *switch TYPE qid cases* 47a⟩+≡ (45b) ◁46d 47c▷

```
  case Qclone:
  case Qstats:
      return ip2gen(c, TYPE(c->qid), dp);
```

Uses Qclone-224 34i, Qstats-225 63c, TYPE-245 35e, and ip2gen() 47b.

⟨*function* ip2gen 47b⟩≡ (331c)

```
  static int
  ip2gen(Chan *c, int i, Dir *dp)
  {
      Qid q;

      switch(i) {
      case Qclone:
          mkqid(&q, QID(PROTO(c->qid), 0, Qclone), 0, QTFILE);
          devdir(c, q, "clone", 0, network, 0666, dp);
          return 1;
      case Qstats:
          mkqid(&q, QID(PROTO(c->qid), 0, Qstats), 0, QTFILE);
          devdir(c, q, "stats", 0, network, 0444, dp);
          return 1;
      }
      return -1;
  }
```

Uses PROTO-247 35c, QID-248 35b, Qclone-224 34i, Qstats-225 63c, and network-249 46a.

/net/proto/conv/

⟨ipgen() *switch TYPE qid cases* 47c⟩+≡ (45b) ◁47a 47d▷

```
  case Qconvdir:
      if(s == DEVDOTDOT){
          s = PROTO(c->qid);
          mkqid(&q, QID(s, 0, Qprotodir), 0, QTDIR);
          devdir(c, q, f->p[s]->name, 0, network, 0555, dp);
          return 1;
      }
      return ip3gen(c, s+Qconvbase, dp);
```

Uses PROTO-247 35c, QID-248 35b, Qconvbase-227 34i, Qconvdir-226 34i, Qprotodir-222 34i, ip3gen() 48a, and network-249 46a.

/net/proto/conv/xxx

⟨ipgen() *switch TYPE qid cases* 47d⟩+≡ (45b) ◁47c

```
  case Qctl:
  case Qdata:
```

```
case Qerr:
case Qlisten:
case Qlocal:
case Qremote:
case Qstatus:
case Qsnoop:
    return ip3gen(c, TYPE(c->qid), dp);
```

Uses Qctl-228 34i, Qdata-229 34i, Qerr-230 62b, Qlisten-231 62e, Qlocal-233 64h, Qremote-234 64h, Qsnoop-235 192a, Qstatus-232 64d, TYPE-245 35e, and ip3gen() 48a.

⟨*function* ip3gen 48a⟩≡ (331c)

```
static int
ip3gen(Chan *c, int i, Dir *dp)
{
    Qid q;
    Conv *cv;
    char *p;

    cv = ipfs[c->dev]->p[PROTO(c->qid)]->conv[CONV(c->qid)];
    if(cv->owner == nil)
        kstrdup(&cv->owner, eve);
    mkqid(&q, QID(PROTO(c->qid), CONV(c->qid), i), 0, QTFILE);
    switch(i) {
    ⟨ip3gen() switch TYPE qid cases 48b⟩
    default:
        return -1;
    }
    devdir(c, q, p, 0, cv->owner, 0444, dp);
    return 1;
}
```

Uses CONV-246 35d, PROTO-247 35c, QID-248 35b, and ipfs 29f.

⟨ip3gen() *switch TYPE qid cases* 48b⟩≡ (48a) 62c ▷

```
case Qctl:
    devdir(c, q, "ctl", 0, cv->owner, cv->perm, dp);
    return 1;
case Qdata:
    devdir(c, q, "data", qlen(cv->rq), cv->owner, cv->perm, dp);
    return 1;
```

Uses Qctl-228 34i and Qdata-229 34i.

5.1.2 Dispatch functions, ipxxx()

The ipopen, ipclose, ipread, and ipwrite functions are the kernel device methods that dispatch file operations to the appropriate protocol or conversation. They all follow the same pattern: extract the protocol index and conversation number from the channel's Qid, look up the Proto and Conv, and switch on the file type to decide what to do.

ipopen()

⟨*global* m2p 48c⟩≡ (331c)

```
static int m2p[] = {
    [OREAD]     4,
    [OWRITE]    2,
    [ORDWR]     6
};
```

⟨*function* ipopen 49a⟩≡ (331c)

```
static Chan*
ipopen(Chan* c, int omode)
{
    Fs *f;
    int perm;
    Proto *p;
    Conv *cv, *nc;

    f = ipfs[c->dev];
    perm = m2p[omode&3];
    switch(TYPE(c->qid)) {
    ⟨ipopen() switch TYPE qid cases 49b⟩
    default:
        break;
    }
    c->mode = openmode(omode);
    c->flag |= COPEN;
    c->offset = 0;
    return c;
}
```

Uses TYPE-245 35e, ipfs 29f, and m2p-251 48c.

⟨ipopen() *switch TYPE qid cases* 49b⟩≡ (49a) 49c ▷

```
case Qtopdir:
case Qprotodir:
case Qconvdir:

case Qipselftab:

case Qstatus:
case Qlocal:
case Qremote:
case Qstats:
    if(omode != OREAD)
        error(Eperm);
    break;
```

Uses Qconvdir-226 34i, Qipselftab-219 130a, Qlocal-233 64h, Qprotodir-222 34i, Qremote-234 64h, Qstats-225 63c, Qstatus-232 64d, and Qtopdir-215 34i.

⟨ipopen() *switch TYPE qid cases* 49c⟩+≡ (49a) ◁ 49b 52a ▷

```
case Qctl:
case Qdata:
case Qerr:
    p = f->p[PROTO(c->qid)];
    qlock(p);
    cv = p->conv[CONV(c->qid)];
    qlock(cv);
    if(waserror()) {
        qunlock(cv);
        qunlock(p);
        nexterror();
    }
    if((perm & (cv->perm>>6)) != perm) {
        if(strcmp(ATTACHER(c), cv->owner) != 0)
            error(Eperm);
        if((perm & cv->perm) != perm)
            error(Eperm);

    }
```

```
    cv->inuse++;
    if(cv->inuse == 1){
        kstrdup(&cv->owner, ATTACHER(c));
        cv->perm = 0660;
    }
    qunlock(cv);
    qunlock(p);
    poperror();
    break;
```

Uses ATTACHER-250 41b, CONV-246 35d, PROTO-247 35c, Qctl-228 34i, Qdata-229 34i, and Qerr-230 62b.

ipclose()

⟨*function* ipclose 50a⟩≡ (331c)

```
static void
ipclose(Chan* c)
{
    Fs *f;

    f = ipfs[c->dev];
    switch(TYPE(c->qid)) {
    ⟨ipclose() switch TYPE qid cases 50b⟩
    default:
        break;
    }
    free(((IPaux*)c->aux)->owner);
    free(c->aux);
}
```

Uses TYPE-245 35e and ipfs 29f.

⟨ipclose() *switch TYPE qid cases* 50b⟩≡ (50a) 189g ▷

```
case Qctl:
case Qdata:
case Qerr:
    if(c->flag & COPEN)
        closeconv(f->p[PROTO(c->qid)]->conv[CONV(c->qid)]);
    break;
```

Uses CONV-246 35d, PROTO-247 35c, Qctl-228 34i, Qdata-229 34i, Qerr-230 62b, and closeconv() 50c.

⟨*function* closeconv 50c⟩≡ (331c)

```
void
closeconv(Conv *cv)
{
    Conv *nc;
    ⟨closeconv() locals 181b⟩

    qlock(cv);

    if(--cv->inuse > 0) {
        qunlock(cv);
        return;
    }
    ⟨closeconv() close incoming calls 63b⟩

    kstrdup(&cv->owner, network);
    cv->perm = 0660;
    ⟨closeconv() if multi, call ipifcremmulti 181c⟩
    cv->r = nil;
    cv->rgen = 0;
```

```
    // Protocol dispatch
    cv->p->close(cv);
    cv->state = Idle;

    qunlock(cv);
}
```

Uses Idle 34b and network-249 46a.

ipread()

⟨*function* ipread 51a⟩≡ (331c)

```
static long
ipread(Chan *ch, void *a, long n, vlong off)
{
    Fs *f;
    Proto *x;
    Conv *cv;
    char *buf, *p;
    long rv;
    ulong offset = off;

    f = ipfs[ch->dev];
    p = a;
    switch(TYPE(ch->qid)) {
    ⟨ipread() switch TYPE qid cases 51b⟩
    default:
        error(Eperm);
        return -1; // unreachable

    }
}
```

Uses TYPE-245 35e and ipfs 29f.

⟨ipread() *switch TYPE qid cases* 51b⟩≡ (51a) 54d ▷

```
case Qtopdir:
case Qprotodir:
case Qconvdir:
    return devdirread(ch, a, n, 0, 0, ipgen);
```

Uses Qconvdir-226 34i, Qprotodir-222 34i, Qtopdir-215 34i, and ipgen() 45b.

ipwrite()

⟨*function* ipwrite 51c⟩≡ (331c)

```
static long
ipwrite(Chan* ch, void *v, long n, vlong off)
{
    Fs *f;
    Proto *x;
    Conv *cv;
    char *p; // err?
    Cmdbuf *cb;
    char *a;
    ulong offset = off;
    ⟨ipwrite() locals 180b⟩

    f = ipfs[ch->dev];
    a = v;
    switch(TYPE(ch->qid)){
```

```
    ⟨ipwrite() switch TYPE qid cases 55a⟩
    default:
        error(Eperm);
    }
    return n;
}
```
Uses TYPE-245 35e and ipfs 29f.

5.1.3 /net/x/clone

Opening `/net/tcp/clone` allocates a new conversation for the protocol. `Fsprotoclone` scans the protocol's `conv` array for a free or reusable slot, initializes it with blank addresses and idle state, and calls the protocol's `create` method to set up the read/write queues. The returned channel's Qid is rewritten to point to the new conversation's `ctl` file—so the caller can immediately write "connect" or "announce" commands to it.

⟨*ipopen() switch TYPE qid cases* 52a⟩+≡ (49a) ◁49c 62g▷

```
case Qclone:
    p = f->p[PROTO(c->qid)];

    qlock(p);
    if(waserror()){
        qunlock(p);
        nexterror();
    }

    cv = Fsprotoclone(p, ATTACHER(c));

    qunlock(p);
    poperror();
    if(cv == nil) {
        error(Enodev);
        break;
    }
    mkqid(&c->qid, QID(p->x, cv->x, Qctl), 0, QTFILE);
    break;
```
Uses ATTACHER-250 41b, Fsprotoclone() 52b, PROTO-247 35c, QID-248 35b, Qclone-224 34i, and Qctl-228 34i.

⟨*function* Fsprotoclone 52b⟩≡ (331c)

```
/*
 *  called with protocol locked
 */
Conv*
Fsprotoclone(Proto *p, char *user)
{
    Conv *cv, **pp, **ep;

retry:
    cv = nil;
    ⟨Fsprotoclone() finding an available conversation in the protocol 53a⟩
    ⟨Fsprotoclone() if no more available conv, garbage collect and retry 54b⟩

    cv->inuse = 1;

    kstrdup(&cv->owner, user);
    cv->perm = 0660;

    cv->state = Idle;
```

```
    ipmove(cv->laddr, IPnoaddr);
    ipmove(cv->raddr, IPnoaddr);
    cv->lport = 0;
    cv->rport = 0;

    cv->r = nil;
    cv->rgen = 0;

    cv->restricted = false;
    cv->maxfragsize = 0;
    cv->ttl = MAXTTL;

    qreopen(cv->rq);
    qreopen(cv->wq);
    qreopen(cv->eq);

    qunlock(cv);
    return cv;
}
```

Uses `IPnoaddr` 23b, `Idle` 34b, `MAXTTL` 81g, and `ipmove` 23c.

Finding a free entry

⟨`Fsprotoclone()` *finding an available conversation in the protocol* 53a⟩≡ (52b)

```
ep = &p->conv[p->nc];
for(pp = p->conv; pp < ep; pp++) {
    cv = *pp;
    // found an unallocated entry in the array
    if(cv == nil){
        cv = malloc(sizeof(Conv));
        if(cv == nil)
            error(Enomem);
        qlock(cv);

        cv->p = p;
        cv->x = pp - p->conv;
        if(p->ptclsize != 0){
            cv->ptcl = malloc(p->ptclsize);
            if(cv->ptcl == nil) {
                free(cv);
                error(Enomem);
            }
        }
        *pp = cv;
        p->ac++;
        cv->eq = qopen(1024, Qmsg, 0, 0);

        // !! Protocol dispatch !!! will create extra queues
        (*p->create)(cv);

        break;
    }
    ⟨Fsprotoclone() if found an unused entry 53b⟩
}
```

Reuse an unused entry

⟨`Fsprotoclone()` *if found an unused entry* 53b⟩≡ (53a)

```
if(canqlock(cv)){
    /*
     *  make sure both processes and protocol
     *  are done with this Conv
     */
    if(cv->inuse == 0 && (p->inuse == nil ||
        // Protocol dispatch
        (*p->inuse)(cv) == false)
      )
        break;

    qunlock(cv);
}
```

⟨Proto(kernel) *conversation methods* 54a⟩+≡ (32f) ◁33a 220▷

```
bool   (*inuse)(Conv*);
```

Garbage collecting

⟨Fsprotoclone() *if no more available conv, garbage collect and retry* 54b⟩≡ (52b)

```
if(pp >= ep) {
    if(p->gc)
        print("Fsprotoclone: garbage collecting Convs\n");
    if(p->gc != nil &&
        // Protocol dispatch
        (*p->gc)(p)
       )
        goto retry;

    /* debugging: do we ever get here? */
    if (cpuserver)
        panic("Fsprotoclone: all conversations in use");
    return nil;
}
```

⟨Proto(kernel) *protocol methods* 54c⟩+≡ (32f) ◁32g 64a▷

```
int   (*gc)(Proto*);  /* returns true if any conversations are freed */
```

5.1.4 /net/x/y/ctl

Reading `ctl` returns the conversation number as a string. Writing to `ctl` dispatches commands: "connect 10.0.0.1!80" calls `connectctlmsg` which sets the remote address/port and sleeps until the connection is established; "announce *!80" calls `announcectlmsg` which prepares the conversation to accept incoming connections. `Fsstdconnect` parses the "ip!port" address string into the conversation's `raddr`/`rport` fields and looks up a route to find the local address.

⟨ipread() *switch TYPE qid cases* 54d⟩+≡ (51a) ◁51b 61d▷

```
case Qctl:
    buf = smalloc(16);
    snprint(buf, 16, "%lud", CONV(ch->qid));
    rv = readstr(offset, p, n, buf);
    free(buf);
    return rv;
```

Uses `CONV`-246 35d, `Qctl`-228 34i, and `readstr()`.

⟨ipwrite() *switch TYPE qid cases* 55a⟩≡ (51c) 61e▷

```
case Qctl:
    x = f->p[PROTO(ch->qid)];
    cv = x->conv[CONV(ch->qid)];
    cb = parsecmd(a, n);

    qlock(cv);
    if(waserror()) {
        qunlock(cv);
        free(cb);
        nexterror();
    }
    if(cb->nf < 1)
        error("short control request");

    ⟨ipwrite() Qctl case, if connect string 55b⟩
    ⟨ipwrite() Qctl case, else if announce string 57b⟩
    ⟨ipwrite() Qctl case, else if other string 61a⟩
    else if(x->ctl != nil) {
        // Protocol dispatch
        p = x->ctl(cv, cb->f, cb->nf);
        if(p != nil)
            error(p);
    } else
        error("unknown control request");
    qunlock(cv);
    free(cb);
    poperror();
    break;
```

Uses `CONV`-246 35d, `PROTO`-247 35c, `Qctl`-228 34i, and `parsecmd()`.

Connect

⟨ipwrite() *Qctl case, if connect string* 55b⟩≡ (55a)

```
if(strcmp(cb->f[0], "connect") == 0)
    connectctlmsg(x, cv, cb);
```

Uses `connectctlmsg()` 55c.

⟨*function* connectctlmsg 55c⟩≡ (331c)

```
static void
connectctlmsg(Proto *p, Conv *c, Cmdbuf *cb)
{
    char *err;

    if(c->state != Idle)
        error(Econinuse);

    c->state = Connecting;
    c->cerr[0] = '\0';

    if(p->connect == nil)
        error("connect not supported");
    // Protocol dispatch
    err = p->connect(c, cb->f, cb->nf);
    if(err != nil)
        error(err);

    qunlock(c);
    if(waserror()){
```

```
        qlock(c);
        nexterror();
    }
    sleep(&c->cr, connected, c);
    qlock(c);
    poperror();

    if(c->cerr[0] != '\0')
        error(c->cerr);
}
```

Uses `Connecting` 34b, `Idle` 34b, and `connected()` 56a.

⟨*function* `connected` 56a⟩≡ (331c)

```
/*
 *  initiate connection and sleep till its set up
 */
static bool
connected(void* a)
{
    return ((Conv*)a)->state == Connected;
}
```

Uses `Connected` 34b.

⟨*function* `Fsstdconnect` 56b⟩≡ (331c)

```
/*
 *  called by protocol connect routine to set addresses
 */
char*
Fsstdconnect(Conv *c, char *argv[], int argc)
{
    char *err;

    switch(argc) {
    case 2:
        err = setraddrport(c, argv[1]);
        if(err != nil)
            return err;
        setladdr(c);
        err = setlport(c);
        if (err != nil)
            return err;
        break;
    case 3:
        err = setraddrport(c, argv[1]);
        if(err != nil)
            return err;
        err = setladdrport(c, argv[2], 0);
        if(err != nil)
            return err;
    default:
        return "bad args to connect";
    }

    ⟨Fsstdconnect() set ipversion field to V4 or V6 516f⟩

    return nil;
}
```

Uses `setladdr()` 329c, `setladdrport()` 59, `setlport()` 58f, and `setraddrport()` 60.

⟨*function* Fsconnected 57a⟩≡ (331c)

```
int
Fsconnected(Conv* c, char* msg)
{
    if(msg != nil && *msg != '\0')
        strncpy(c->cerr, msg, ERRMAX-1);

    switch(c->state){
    case Connecting:
        c->state = Connected;
        break;
    ⟨Fsconnected() switch state cases 58c⟩
    }
    wakeup(&c->cr);
    return 0;
}
```

Announce

⟨ipwrite() *Qctl case, else if announce string* 57b⟩≡ (55a)

```
else if(strcmp(cb->f[0], "announce") == 0)
    announcectlmsg(x, cv, cb);
```

Uses announcectlmsg() 57c.

⟨*function* announcectlmsg 57c⟩≡ (331c)

```
static void
announcectlmsg(Proto *p, Conv *c, Cmdbuf *cb)
{
    char *err;

    if(c->state != Idle)
        error(Econinuse);

    c->state = Announcing;
    c->cerr[0] = '\0';

    if(p->announce == nil)
        error("announce not supported");
    // Protocol dispatch
    err = p->announce(c, cb->f, cb->nf);

    if(err != nil)
        error(err);

    qunlock(c);
    if(waserror()){
        qlock(c);
        nexterror();
    }
    sleep(&c->cr, announced, c);
    qlock(c);
    poperror();

    if(c->cerr[0] != '\0')
        error(c->cerr);
}
```

Uses Announcing 34b, Idle 34b, and announced() 58a.

⟨*function* announced 58a⟩≡ (331c)
```
/*
 *  initiate announcement and sleep till its set up
 */
static bool
announced(void* a)
{
    return ((Conv*)a)->state == Announced;
}
```
Uses Announced 34b.

⟨*function* Fsstdannounce 58b⟩≡ (331c)
```
/*
 *  called by protocol announce routine to set addresses
 */
char*
Fsstdannounce(Conv* c, char* argv[], int argc)
{
    memset(c->raddr, 0, sizeof(c->raddr));
    c->rport = 0;
    switch(argc){
    default:
        break;
    case 2:
        return setladdrport(c, argv[1], 1);
    }
    return "bad args to announce";
}
```
Uses setladdrport() 59.

⟨Fsconnected() *switch state cases* 58c⟩≡ (57a)
```
case Announcing:
    c->state = Announced;
    break;
```

Port settings

⟨Conv(kernel) *other fields* 58d⟩+≡ (33e) ◁34h 81a▷
```
bool restricted;   /* remote port is restricted */
```

⟨Proto(kernel) *other fields* 58e⟩+≡ (32a) ◁32d
```
ushort    nextrport;
```

⟨*function* setlport 58f⟩≡ (331c)
```
/*
 *  pick a local port and set it
 */
char *
setlport(Conv* c)
{
    Proto *p;
    int i, port;

    p = c->p;
    qlock(p);
    if(c->restricted){
        /* Restricted ports cycle between 600 and 1024. */
        for(i=0; i<1024-600; i++){
            if(p->nextrport >= 1024 || p->nextrport < 600)
```

```
                p->nextrport = 600;
            port = p->nextrport++;
            if(!lportinuse(p, port))
                goto chosen;
        }
    }else{
        /*
         * Unrestricted ports are chosen randomly
         * between 2^15 and 2^16.  There are at most
         * 4*Nchan = 4096 ports in use at any given time,
         * so even in the worst case, a random probe has a
         * 1 - 4096/2^15 = 87% chance of success.
         * If 64 successive probes fail, there is a bug somewhere
         * (or a once in 10^58 event has happened, but that's
         * less likely than a venti collision).
         */
        for(i=0; i<64; i++){
            port = (1<<15) + nrand(1<<15);
            if(!lportinuse(p, port))
                goto chosen;
        }
    }
    qunlock(p);
    /*
     * debugging: let's see if we ever get this.
     * if we do (and we're a cpu server), we might as well restart
     * since we're now unable to service new connections.
     */
    panic("setlport: out of ports");
    return "no ports available";

chosen:
    c->lport = port;
    qunlock(p);
    return nil;
}
```

Uses lportinuse() 330a.

⟨*function* setladdrport 59⟩≡ (331c)

```
/*
 *  set a local address and port from a string of the form
 *  [address!]port[!r]
 */
char*
setladdrport(Conv* c, char* str, int announcing)
{
    char *p;
    char *rv;
    ushort lport;
    ipaddr addr;

    /*
     *  ignore restricted part if it exists.  it's
     *  meaningless on local ports.
     */
    p = strchr(str, '!');
    if(p != nil){
        *p++ = 0;
        if(strcmp(p, "r") == 0)
            p = nil;
```

```
    }

    c->lport = 0;
    if(p == nil){
        if(announcing)
            ipmove(c->laddr, IPnoaddr);
        else
            setladdr(c);
        p = str;
    } else {
        if(strcmp(str, "*") == 0)
            ipmove(c->laddr, IPnoaddr);
        else {
            if(parseip(addr, str) == -1)
                return Ebadip;
            if(ipforme(c->p->f, addr))
                ipmove(c->laddr, addr);
            else
                return "not a local IP address";
        }
    }

    /* one process can get all connections */
    if(announcing && strcmp(p, "*") == 0){
        if(!iseve())
            error(Eperm);
        return setluniqueport(c, 0);
    }

    lport = atoi(p);
    if(lport <= 0)
        rv = setlport(c);
    else
        rv = setluniqueport(c, lport);
    return rv;
}
```

Uses `IPnoaddr` 23b, `ipforme()` 125e, `ipmove` 23c, `parseip()` 89, `setladdr()` 329c, `setlport()` 58f, and `setluniqueport()` 329d.

⟨*function* `setraddrport` 60⟩≡ (331c)

```
static char*
setraddrport(Conv* c, char* str)
{
    char *p;

    p = strchr(str, '!');
    if(p == nil)
        return "malformed address";
    *p++ = 0;
    if (parseip(c->raddr, str) == -1)
        return Ebadip;
    c->rport = atoi(p);
    p = strchr(p, '!');
    if(p){
        if(strstr(p, "!r") != nil)
            c->restricted = true;
    }
    return nil;
}
```

Uses `parseip()` 89.

Bind

⟨ipwrite() *Qctl case, else if other string* 61a⟩≡ (55a) 81b ▷
```
  else if(strcmp(cb->f[0], "bind") == 0)
      bindctlmsg(x, cv, cb);
```
Uses bindctlmsg() 61b.

⟨*function* bindctlmsg 61b⟩≡ (331c)
```
  static void
  bindctlmsg(Proto *x, Conv *cv, Cmdbuf *cb)
  {
      char *p;

      if(x->bind == nil)
          p = Fsstdbind(cv, cb->f, cb->nf);
      else
          // Protocol dispatch
          p = x->bind(cv, cb->f, cb->nf);
      if(p != nil)
          error(p);
  }
```
Uses Fsstdbind() 61c.

⟨*function* Fsstdbind 61c⟩≡ (331c)
```
  /*
   *  called by protocol bind routine to set addresses
   */
  char*
  Fsstdbind(Conv* cv, char* argv[], int argc)
  {
      switch(argc){
      default:
          break;
      case 2:
          return setladdrport(cv, argv[1], 0);
      }
      return "bad args to bind";
  }
```
Uses setladdrport() 59.

5.1.5 /net/x/y/data

The `data` file is the actual data channel. Reading dequeues bytes from the conversation's receive queue (`qread(cv->rq, ...)`), and writing enqueues bytes into the write queue (`qwrite(cv->wq, ...)`). The protocol's "kick" function (installed by `create`) is called automatically when data is written, triggering the protocol to add headers and transmit the packet.

⟨ipread() *switch TYPE qid cases* 61d⟩+≡ (51a) ◁ 54d 62d ▷
```
  case Qdata:
      cv = f->p[PROTO(ch->qid)]->conv[CONV(ch->qid)];
      return qread(cv->rq, a, n);
```
Uses CONV-246 35d, PROTO-247 35c, and Qdata-229 34i.

⟨ipwrite() *switch TYPE qid cases* 61e⟩+≡ (51c) ◁ 55a 109a ▷
```
  case Qdata:
      cv = f->p[PROTO(ch->qid)]->conv[CONV(ch->qid)];
      if(cv->wq == nil)
          error(Eperm);
```

```
    qwrite(cv->wq, a, n);
    break;
```
Uses CONV-246 35d, PROTO-247 35c, and Qdata-229 34i.

5.1.6 /net/x/y/err

⟨Conv(kernel) *queue fields* 62a⟩+≡ (33e) ◁34d
```
 Queue*  eq;     /* returned error packets */
```

⟨Qid *conversation extra cases* 62b⟩≡ (34i) 62e▷
```
 Qerr,
```

⟨ip3gen() *switch TYPE qid cases* 62c⟩+≡ (48a) ◁48b 62f▷
```
 case Qerr:
     devdir(c, q, "err", qlen(cv->eq), cv->owner, cv->perm, dp);
     return 1;
```
Uses Qerr-230 62b.

⟨ipread() *switch TYPE qid cases* 62d⟩+≡ (51a) ◁61d 64b▷
```
 case Qerr:
     cv = f->p[PROTO(ch->qid)]->conv[CONV(ch->qid)];
     return qread(cv->eq, a, n);
```
Uses CONV-246 35d, PROTO-247 35c, and Qerr-230 62b.

5.1.7 /net/x/y/listen

Opening the `listen` file blocks until a remote machine connects. The caller must have already announced on this conversation. `sleep` waits on `cv->listenr` until `incoming` returns true (meaning `cv->incall` is non-nil). When a connection arrives, the protocol's `rcv` handler creates a new `Conv` and chains it on `cv->incall`; the `listen` opener dequeues it and rewrites its Qid to the new conversation's `ctl`.

⟨Qid *conversation extra cases* 62e⟩+≡ (34i) ◁62b 64d▷
```
 Qlisten,
```

⟨ip3gen() *switch TYPE qid cases* 62f⟩+≡ (48a) ◁62c 64e▷
```
 case Qlisten:
     devdir(c, q, "listen", 0, cv->owner, cv->perm, dp);
     return 1;
```
Uses Qlisten-231 62e.

⟨ipopen() *switch TYPE qid cases* 62g⟩+≡ (49a) ◁52a 108b▷
```
 case Qlisten:
     cv = f->p[PROTO(c->qid)]->conv[CONV(c->qid)];
     if((perm & (cv->perm>>6)) != perm) {
         if(strcmp(ATTACHER(c), cv->owner) != 0)
             error(Eperm);
         if((perm & cv->perm) != perm)
             error(Eperm);

     }

     if(cv->state != Announced)
         error("not announced");

     if(waserror()){
         closeconv(cv);
         nexterror();
```

```
    }
    qlock(cv);
    cv->inuse++;
    qunlock(cv);

    nc = nil;
    while(nc == nil) {
        /* give up if we got a hangup */
        if(qisclosed(cv->rq))
            error("listen hungup");

        qlock(&cv->listenq);
        if(waserror()) {
            qunlock(&cv->listenq);
            nexterror();
        }

        /* wait for a connect */
        sleep(&cv->listenr, incoming, cv);

        qlock(cv);
        nc = cv->incall;
        if(nc != nil){
            cv->incall = nc->next;
            mkqid(&c->qid, QID(PROTO(c->qid), nc->x, Qctl), 0, QTFILE);
            kstrdup(&cv->owner, ATTACHER(c));
        }
        qunlock(cv);

        qunlock(&cv->listenq);
        poperror();
    }
    closeconv(cv);
    poperror();
    break;
```

Uses `ATTACHER-250` 41b, `Announced` 34b, `CONV-246` 35d, `PROTO-247` 35c, `QID-248` 35b, `Qctl-228` 34i, `Qlisten-231` 62e, `closeconv()` 50c, and `incoming()` 328b.

⟨`Conv(kernel)` *listen fields* 63a⟩≡ (33e)

```
Conv* incall;     /* calls waiting to be listened for */
Conv* next;
QLock listenq;
Rendez  listenr;
```

⟨`closeconv()` *close incoming calls* 63b⟩≡ (50c)

```
/* close all incoming calls since no listen will ever happen */
for(nc = cv->incall; nc; nc = cv->incall){
    cv->incall = nc->next;
    closeconv(nc);
}
cv->incall = nil;
```

Uses `closeconv()` 50c.

5.1.8 Other files

`/net/x/stats`

⟨`Qid` *protocol extra cases* 63c⟩≡ (34i)

```
Qstats,
```

⟨Proto(kernel) *protocol methods* 64a⟩+≡ (32f) ◁54c

```
  int   (*stats)(Proto*, char*, int);
```

⟨ipread() *switch TYPE qid cases* 64b⟩+≡ (51a) ◁62d 64g▷

```
  case Qstats:
      x = f->p[PROTO(ch->qid)];
      if(x->stats == nil)
          error("stats not implemented");
      buf = smalloc(Statelen);

      // Protocol dispatch
      (*x->stats)(x, buf, Statelen);

      rv = readstr(offset, p, n, buf);
      free(buf);
      return rv;
```

Uses PROTO-247 35c, Qstats-225 63c, Statelen-252 64c, and readstr().

⟨*constant* Statelen 64c⟩≡ (329a)

```
      Statelen=   32*1024,
```

/net/x/y/status

⟨Qid *conversation extra cases* 64d⟩+≡ (34i) ◁62e 64h▷

```
  Qstatus,
```

⟨ip3gen() *switch TYPE qid cases* 64e⟩+≡ (48a) ◁62f 65a▷

```
  case Qstatus:
      p = "status";
      break;
```

Uses Qstatus-232 64d.

⟨Proto(kernel) *conversation inspection methods* 64f⟩≡ (32f) 65b▷

```
  int   (*state)(Conv*, char*, int);
```

⟨ipread() *switch TYPE qid cases* 64g⟩+≡ (51a) ◁64b 65c▷

```
  case Qstatus:
      buf = smalloc(Statelen);
      x = f->p[PROTO(ch->qid)];
      cv = x->conv[CONV(ch->qid)];

      // Protocol dispatch
      (*x->state)(cv, buf, Statelen-2);

      rv = readstr(offset, p, n, buf);
      free(buf);
      return rv;
```

Uses CONV-246 35d, PROTO-247 35c, Qstatus-232 64d, Statelen-252 64c, and readstr().

/net/x/y/local, /net/x/y/remote

Reading `local` returns the local IP address and port of the conversation as a string (e.g., `10.0.2.15!1234`); `remote` returns the remote side. These are used by servers to identify who connected to them.

⟨Qid *conversation extra cases* 64h⟩+≡ (34i) ◁64d

```
  Qlocal,
  Qremote,
```

⟨ip3gen() *switch TYPE qid cases* 65a⟩+≡ (48a) ◁64e 193a▷
```
  case Qlocal:
      p = "local";
      break;
  case Qremote:
      p = "remote";
      break;
```
Uses `Qlocal-233` 64h and `Qremote-234` 64h.

⟨Proto(kernel) *conversation inspection methods* 65b⟩+≡ (32f) ◁64f 65d▷
```
  int   (*local)(Conv*, char*, int);
```

⟨ipread() *switch TYPE qid cases* 65c⟩+≡ (51a) ◁64g 65e▷
```
  case Qlocal:
      buf = smalloc(Statelen);
      x = f->p[PROTO(ch->qid)];
      cv = x->conv[CONV(ch->qid)];
      if(x->local == nil) {
          snprint(buf, Statelen, "%I!%d\n", cv->laddr, cv->lport);
      } else {
          // Protocol dispatch
          (*x->local)(cv, buf, Statelen-2);
      }
      rv = readstr(offset, p, n, buf);
      free(buf);
      return rv;
```
Uses `CONV-246` 35d, `PROTO-247` 35c, `Qlocal-233` 64h, `Statelen-252` 64c, and `readstr()`.

⟨Proto(kernel) *conversation inspection methods* 65d⟩+≡ (32f) ◁65b
```
  int   (*remote)(Conv*, char*, int);
```

⟨ipread() *switch TYPE qid cases* 65e⟩+≡ (51a) ◁65c 108c▷
```
  case Qremote:
      buf = smalloc(Statelen);
      x = f->p[PROTO(ch->qid)];
      cv = x->conv[CONV(ch->qid)];
      if(x->remote == nil) {
          snprint(buf, Statelen, "%I!%d\n", cv->raddr, cv->rport);
      } else {
          // Protocol dispatch
          (*x->remote)(cv, buf, Statelen-2);
      }
      rv = readstr(offset, p, n, buf);
      free(buf);
      return rv;
```
Uses `CONV-246` 35d, `PROTO-247` 35c, `Qremote-234` 64h, `Statelen-252` 64c, and `readstr()`.

5.2 Ethernet device

⟨*global* etherdevtab 65f⟩≡ (264b)
```
  Dev etherdevtab = {
      .dc       =    'l',
      .name     =    "ether",

      .attach   =    etherattach,
      .walk     =    etherwalk,
      .open     =    etheropen,
      .close    =    etherclose,
```

```
    .read     =    etherread,
    .write    =    etherwrite,
    .stat     =    etherstat,
    .wstat    =    etherwstat,

    .reset    =    etherreset,
    .init     =    devinit,
    .shutdown =    ethershutdown,
    .create   =    ethercreate,
    .bread    =    etherbread,
    .bwrite   =    etherbwrite,
    .remove   =    devremove,
};
```

5.2.1 `/net/etherx` hierarchy and `etherwalk()`

Chapter 6

Configuration

Network configuration in Plan 9 is done through the `/net/ipifc` pseudo-protocol—it reuses the same `Proto`/`Conv` mechanism as real protocols, but instead of carrying data, its `ctl` commands configure IP interfaces. Writing `bind ether /net/ether0` to `/net/ipifc/0/ctl` associates an Ethernet device with an IP interface, and `add 10.0.0.2` assigns an IP address. This elegant reuse of the protocol abstraction avoids introducing a separate configuration mechanism.

6.1 /net/ipifc/ protocol

6.1.1 Protocol initialisation

`ipifcinit` registers `ipifc` as a protocol in the `Fs` stack. Unlike transport protocols (UDP, TCP), `ipifc` has no IP protocol number—it is used purely for configuration. Its `ctl` method dispatches "bind", "add", "remove", and "unbind" commands to configure the network stack.

⟨*function* `ipifcinit` 67⟩≡ (384b)

```
void
ipifcinit(Fs *f)
{
    Proto *ipifc;

    ipifc = smalloc(sizeof(Proto));

    ipifc->name = "ipifc";
    ipifc->create = ipifccreate;
    ipifc->close = ipifcclose;

    ipifc->bind = ipifcbind;
    ipifc->connect = ipifcconnect;
    ipifc->announce = nil;
    ipifc->ctl = ipifcctl;

    ipifc->rcv = nil;
    ipifc->advise = nil;
    ipifc->inuse = ipifcinuse;

    ipifc->local = ipifclocal;
    ipifc->state = ipifcstate;
    ipifc->stats = ipifcstats;

    ipifc->ipproto = -1;

    ipifc->nc = Maxmedia;
```

```
    ipifc->ptclsize = sizeof(Ipifc);

    ⟨ipifcinit() modify f 68c⟩

    Fsproto(f, ipifc);
}
```
Uses Fsproto() 68a, Maxmedia-98 27b, ipifcbind() 69, ipifcclose() 84b, ipifcconnect() 376c, ipifccreate() 68d, ipifcctl() 71a, ipifcinuse() 376a, ipifclocal() 375d, and ipifcstats() 77a.

⟨*function* Fsproto 68a⟩≡ (331c)
```
int
Fsproto(Fs *f, Proto *p)
{
    if(f->np >= Maxproto)
        return -1;

    p->f = f;

    ⟨Fsproto() adjust f-¿t2p 68b⟩

    p->conv = malloc(sizeof(Conv*) * (p->nc+1));
    if(p->conv == nil)
        panic("Fsproto");

    p->nextrport = 600;

    p->x = f->np;
    f->p[f->np++] = p;

    return 0;
}
```
Uses Maxproto 29e.

⟨Fsproto() *adjust f-¿t2p* 68b⟩≡ (68a)
```
if(p->ipproto > 0){
    if(f->t2p[p->ipproto] != nil)
        return -1;
    f->t2p[p->ipproto] = p;
}
```

⟨ipifcinit() *modify f* 68c⟩≡ (67) 126b▷
```
f->ipifc = ipifc;   /* hack for ipifcremroute, findipifc, ... */
```

6.1.2 /net/ipifc/clone

⟨*function* ipifccreate 68d⟩≡ (384b)
```
/*
 *  called when a new ipifc structure is created
 */
static void
ipifccreate(Conv *cv)
{
    Ipifc *ifc;

    cv->rq = qopen(QMAX, 0, 0, 0);
    cv->wq = qopen(QMAX, Qkick, ipifckick, cv);

    cv->sq = qopen(QMAX, 0, 0, 0);
```

```
    ifc = (Ipifc*)cv->ptcl;
    ifc->m = nil;

    ifc->conv = cv;
    ifc->reassemble = false;
}
```

Uses QMAX-102 374 and ipifckick() 376b.

6.1.3 Binding medium: /net/ipifc/x/ctl bind

Writing `bind ether /net/ether0` to `/net/ipifc/x/ctl` calls `ipifcbind`, which looks up "ether" in the `media` table, opens the Ethernet device via `chandial`, and calls the medium's `bind` method (`etherbind`). This connects the IP interface to the physical network: packets written to `/net/ipifc/x/data` will now go through the Ethernet medium.

⟨*function* ipifcbind 69⟩≡ (384b)

```
/*
 *  attach a device (or pkt driver) to the interface.
 *  called with cv locked
 */
static char*
ipifcbind(Conv *cv, char **argv, int argc)
{
    Ipifc *ifc;
    Medium *m;

    if(argc < 2)
        return Ebadarg;

    ifc = (Ipifc*)cv->ptcl;

    /* bind the device to the interface */
    m = ipfindmedium(argv[1]);
    if(m == nil)
        return "unknown interface type";

    wlock(ifc);
    if(ifc->m != nil){
        wunlock(ifc);
        return "interface already bound";
    }
    if(waserror()){
        wunlock(ifc);
        nexterror();
    }

    // This time Medium dispatch
    /* do medium specific binding */
    (*m->bind)(ifc, argc, argv);

    /* set the bound device name */
    if(argc > 2)
        strncpy(ifc->dev, argv[2], sizeof(ifc->dev));
    else
        snprint(ifc->dev, sizeof ifc->dev, "%s%d", m->name, cv->x);
    ifc->dev[sizeof(ifc->dev)-1] = 0;

    /* set up parameters */
```

```
    ifc->m = m;

    ifc->mintu = ifc->m->mintu;
    ifc->maxtu = ifc->m->maxtu;
    if(ifc->m->unbindonclose == false)
        ifc->conv->inuse++;

    ifc->rp.mflag = 0;      /* default not managed */
    ifc->rp.oflag = 0;
    ifc->rp.maxraint = 600000;  /* millisecs */
    ifc->rp.minraint = 200000;
    ifc->rp.linkmtu = 0;        /* no mtu sent */
    ifc->rp.reachtime = 0;
    ifc->rp.rxmitra = 0;
    ifc->rp.ttl = MAXTTL;
    ifc->rp.routerlt = 3 * ifc->rp.maxraint;

    /* any ancillary structures (like routes) no longer pertain */
    ifc->ifcid++;

    /* reopen all the queues closed by a previous unbind */
    qreopen(cv->rq);
    qreopen(cv->eq);
    qreopen(cv->sq);

    wunlock(ifc);
    poperror();

    return nil;
}
```

Uses `MAXTTL` 81g and `ipfindmedium()` 70c.

⟨`Ipifc(kernel)` *other fields* 70a⟩+≡ (25b) ◁25d 105b▷

```
uchar ifcid;    /* incremented each 'bind/unbind/add/remove' */
```

⟨`Routetree` *other fields* 70b⟩+≡ (38b) ◁38d 119a▷

```
uchar ifcid;    /* must match ifc->id */
```

⟨*function* `ipfindmedium` 70c⟩≡ (384b)

```
/*
 *  find the medium with this name
 */
Medium*
ipfindmedium(char *name)
{
    Medium **mp;

    for(mp = media; *mp != nil; mp++)
        if(strcmp((*mp)->name, name) == 0)
            break;
    return *mp;
}
```

Uses `media` 27a.

6.1.4 /net/ipifc/x/ctl

`ipifcctl` is the protocol-specific control handler for `ipifc`. Unlike transport protocols which handle “connect” and “announce”, `ipifc`'s control commands are configuration-oriented: “add” assigns an IP address, “remove”

deletes one, "unbind" detaches the medium, and various others set routing, MTU, TTL, or TOS parameters.

⟨*function* ipifcctl 71a⟩≡ (384b)
```
/*
 *  non-standard control messages.
 *  called with cv->car locked.
 */
static char*
ipifcctl(Conv* cv, char** argv, int argc)
{
    Ipifc *ifc;
    int i;

    ifc = (Ipifc*)cv->ptcl;
    ⟨ipifcctl() if add string 71b⟩
    ⟨ipifcctl() else if other string 79⟩

    return "unsupported ctl";
}
```

6.1.5 Adding an IP: /net/ipifc/x/ctl add

Writing `add 10.0.2.15 255.255.255.0` to `/net/ipifc/x/ctl` calls `ipifcadd`, which creates an `Iplifc` node, links it into the interface's list, and adds the corresponding routes. This is the step that gives the machine its identity on the network—after this, the IP stack knows which addresses belong to it and can accept incoming packets for them.

⟨ipifcctl() *if add string* 71b⟩≡ (71a)
```
if(strcmp(argv[0], "add") == 0)
    return ipifcadd(ifc, argv, argc, false, nil);
```
Uses ipifcadd() 71c.

⟨*function* ipifcadd 71c⟩≡ (384b)
```
/*
 *  add an address to an interface.
 */
char*
ipifcadd(Ipifc *ifc, char **argv, int argc, bool tentative, Iplifc *lifcp)
{
    int i;
    int mtu;
    // enum<route_type>
    int type;
    ipaddr ip;
    ipaddr mask;
    ipaddr net; // ip & mask
    ipaddr rem;
    Iplifc *lifc, **l;
    Fs *f;

    ⟨ipifcadd() locals 178c⟩

    if(ifc->m == nil)
        return "ipifc not yet bound to device";

    f = ifc->conv->p->f;

    type = Rifc;
    memset(ip, 0, IPaddrlen);
```

```
    memset(mask, 0, IPaddrlen);
    memset(rem, 0, IPaddrlen);

    switch(argc){
    ⟨ipifcadd() switch argc, proxy case, and fall through 182f⟩
    ⟨ipifcadd() switch argc, mtu setting case, and fall through 80b⟩
    ⟨ipifcadd() switch argc cases, setting ip, mask, net, rem 72⟩
    default:
        return Ebadarg;
    }

    ⟨ipifcadd() set tentative for ipv6 516k⟩
    wlock(ifc);
    ⟨ipifcadd() check if already a local address for this ifc 73b⟩

    /* add the address to the list of logical ifc's for this ifc */
    lifc = smalloc(sizeof(Iplifc));
    ipmove(lifc->local, ip);
    ipmove(lifc->mask, mask);
    ipmove(lifc->remote, rem);
    ipmove(lifc->net, net);
    ⟨ipifcadd() set ipv6 fields for lifc 517g⟩
    // add_tail(lifc, ifc->lifc)
    lifc->next = nil;
    for(l = &ifc->lifc; *l; l = &(*l)->next)
        ;
    *l = lifc;

    ⟨ipifcadd() check for point to point interface 182d⟩

    /* add local routes */
    if(isv4(ip))
        v4addroute(f, tifc, rem+IPv4off, mask+IPv4off, rem+IPv4off, type);
    ⟨ipifcadd() add route if ipv6 case 516g⟩

    addselfcache(f, ifc, lifc, ip, Runi);

    ⟨ipifcadd() register proxy if point to point interface or proxy 182e⟩

    if(isv4(ip) || ipcmp(ip, IPnoaddr) == 0) {
        ⟨ipifcadd() add broadcast addresses to self cache 179a⟩
    }
    ⟨ipifcadd() if ipv6 add multicast addresses to self cache 517f⟩

    /* register the address on this network for address resolution */
    if(isv4(ip) && ifc->m->areg != nil)
        // Medium dispatch
        (*ifc->m->areg)(ifc, ip);

out:
    wunlock(ifc);
    ⟨ipifcadd() if ipv6 tentative and broacast 517d⟩
    return nil;
}
```

Uses `IPaddrlen` 20c, `IPnoaddr` 23b, `IPv4off` 21a, `Rifc` 38e, `Runi` 38e, `addselfcache()` 127b, `ipcmp` 23d, `ipmove` 23c, `isv4()` 21b, `tifc-104` 73a, and `v4addroute()` 114.

⟨`ipifcadd()` *switch argc cases, setting ip, mask, net, rem* 72⟩≡ (71c)

```
// add <ip> <mask> <rem>
case 4:
```

```
    if (parseip(ip, argv[1]) == -1 || parseip(rem, argv[3]) == -1)
        return Ebadip;
    parseipmask(mask, argv[2]);
    maskip(rem, mask, net);
    break;
// add <ip> <mask>
case 3:
    if (parseip(ip, argv[1]) == -1)
        return Ebadip;
    parseipmask(mask, argv[2]);
    maskip(ip, mask, rem);
    maskip(rem, mask, net);
    break;
// simplest case, add <ip>
case 2:
    if (parseip(ip, argv[1]) == -1)
        return Ebadip;
    memmove(mask, defmask(ip), IPaddrlen);
    maskip(ip, mask, rem);
    maskip(rem, mask, net);
    break;
```

Uses `IPaddrlen` 20c, `defmask()` 22b, `maskip()` 22d, `parseip()` 89, and `parseipmask()` 224d.

⟨*global* `tifc` 73a⟩≡ (384b)

```
static char tifc[] = "ifc ";
```

Uses `tifc-104` 73a.

⟨`ipifcadd()` *check if already a local address for this ifc* 73b⟩≡ (71c)

```
/* ignore if this is already a local address for this ifc */
for(lifc = ifc->lifc; lifc; lifc = lifc->next) {
    if(ipcmp(lifc->local, ip) == 0) {
        ⟨ipifcadd() when already local address for ifc, copy ipv6 fields 518a⟩
        goto out;
    }
}
```

Uses `ipcmp` 23d.

6.2 IP Interface, user side

The user-side IP interface code reads and parses the files under `/net/ipifc/` to build the `Ipifc` and `Iplifc` linked lists. `readipifc` opens each interface directory in turn, reads the status and address files, and returns a linked list of all configured interfaces with their IP addresses, masks, and statistics.

6.2.1 Parsing

⟨*function* `readipifc` 73c⟩≡ (231b)

```
Ipifc*
readipifc(char *net, Ipifc *ifc, int index)
{
    int fd, i, n;
    Dir *dir;
    char directory[128];
    char buf[128];
    Ipifc **l;

    _freeifc(ifc);
```

```
    l = &ifc;
    ifc = nil;

    if(net == nil)
        net = "/net";
    snprint(directory, sizeof(directory), "%s/ipifc", net);

    if(index >= 0){
        snprint(buf, sizeof(buf), "%s/%d/status", directory, index);
        _readipifc(buf, l, index);
    } else {
        fd = open(directory, OREAD);
        if(fd < 0)
            return nil;
        n = dirreadall(fd, &dir);
        close(fd);

        for(i = 0; i < n; i++){
            if(strcmp(dir[i].name, "clone") == 0)
                continue;
            if(strcmp(dir[i].name, "stats") == 0)
                continue;
            snprint(buf, sizeof(buf), "%s/%s/status", directory, dir[i].name);
            l = _readipifc(buf, l, atoi(dir[i].name));
        }
        free(dir);
    }

    return ifc;
}
```

Uses _freeifc() 76a and _readipifc() 74.

⟨*function* _readipifc 74⟩≡ (231b)

```
static Ipifc**
_readipifc(char *file, Ipifc **l, int index)
{
    int i, n, fd, lines;
    char buf[4*1024];
    char *line[32];
    char *f[64];
    Ipifc *ifc, **l0;
    Iplifc *lifc, **ll;

    /* read the file */
    fd = open(file, OREAD);
    if(fd < 0)
        return l;
    n = 0;
    while((i = read(fd, buf+n, sizeof(buf)-1-n)) > 0 && n < sizeof(buf) - 1)
        n += i;
    buf[n] = 0;
    close(fd);

    //if(strncmp(buf, "device", 6) != 0)
    //    return _readoldipifc(buf, l, index);

    /* ignore ifcs with no associated device */
    if(strncmp(buf+6, "  ", 2) == 0)
        return l;
```

```
    /* allocate new interface */
    *l = ifc = mallocz(sizeof(Ipifc), 1);
    if(ifc == nil)
        return l;
    l0 = l;
    l = &ifc->next;
    ifc->index = index;

    lines = getfields(buf, line, nelem(line), 1, "\n");

    /* pick off device specific info(first line) */
    n = tokenize(line[0], f, nelem(f));
    if(n%2 != 0)
        goto lose;
    strncpy(ifc->dev, findfield("device", f, n), sizeof(ifc->dev));
    ifc->dev[sizeof(ifc->dev)-1] = 0;
    if(ifc->dev[0] == 0){
lose:
        free(ifc);
        *l0 = nil;
        return l;
    }
    ifc->mtu          = strtoul(findfield("maxtu", f, n), nil, 10);
    ifc->sendra6      = atoi(findfield("sendra", f, n));
    ifc->recvra6      = atoi(findfield("recvra", f, n));
    ifc->rp.mflag     = atoi(findfield("mflag", f, n));
    ifc->rp.oflag     = atoi(findfield("oflag", f, n));
    ifc->rp.maxraint  = atoi(findfield("maxraint", f, n));
    ifc->rp.minraint  = atoi(findfield("minraint", f, n));
    ifc->rp.linkmtu   = atoi(findfield("linkmtu", f, n));
    ifc->rp.reachtime = atoi(findfield("reachtime", f, n));
    ifc->rp.rxmitra   = atoi(findfield("rxmitra", f, n));
    ifc->rp.ttl       = atoi(findfield("ttl", f, n));
    ifc->rp.routerlt  = atoi(findfield("routerlt", f, n));
    ifc->pktin        = strtoul(findfield("pktin", f, n), nil, 10);
    ifc->pktout       = strtoul(findfield("pktout", f, n), nil, 10);
    ifc->errin        = strtoul(findfield("errin", f, n), nil, 10);
    ifc->errout       = strtoul(findfield("errout", f, n), nil, 10);

    /* now read the addresses */
    ll = &ifc->lifc;
    for(i = 1; i < lines; i++){
        n = tokenize(line[i], f, nelem(f));
        if(n < 5)
            break;

        /* allocate new local address */
        *ll = lifc = mallocz(sizeof(Iplifc), 1);
        ll = &lifc->next;

        parseip(lifc->ip, f[0]);
        parseipmask(lifc->mask, f[1]);
        parseip(lifc->net, f[2]);

        lifc->validlt = strtoul(f[3], nil, 10);
        lifc->preflt = strtoul(f[4], nil, 10);
    }

    return l;
}
```

Uses `findfield()` 76b, `parseip()` 89, and `parseipmask()` 224d.

⟨*function* `_freeifc` 76a⟩≡ (231b)

```
static void
_freeifc(Ipifc *ifc)
{
    Ipifc *next;
    Iplifc *lnext, *lifc;

    if(ifc == nil)
        return;
    for(; ifc; ifc = next){
        next = ifc->next;
        for(lifc = ifc->lifc; lifc; lifc = lnext){
            lnext = lifc->next;
            free(lifc);
        }
        free(ifc);
    }
}
```

⟨*function* `findfield` 76b⟩≡ (231b)

```
static char*
findfield(char *name, char **f, int n)
{
    int i;

    for(i = 0; i < n-1; i++)
        if(strcmp(f[i], name) == 0)
            return f[i+1];
    return "";
}
```

6.2.2 /net/ipifc/x

⟨*function* `myipaddr` 76c⟩≡ (226b)

```
/* find first ip addr that isn't the friggin loopback address
 * unless there are no others
 */
errorneg1
myipaddr(ipaddr ip, char *net)
{
    Ipifc *nifc;
    Iplifc *lifc;
    static Ipifc *ifc;
    ipaddr mynet;

    ifc = readipifc(net, ifc, -1);
    for(nifc = ifc; nifc; nifc = nifc->next)
        for(lifc = nifc->lifc; lifc; lifc = lifc->next){
            maskip(lifc->ip, loopbackmask, mynet);
            if(ipcmp(mynet, loopbacknet) == 0){
                continue;
            }
            if(ipcmp(lifc->ip, IPnoaddr) != 0){
                ipmove(ip, lifc->ip);
                return OK_0;
            }
        }
```

```
    ipmove(ip, IPnoaddr);
    return ERROR_NEG1;
}
```
Uses IPnoaddr 23b, ipcmp 23d, ipmove 23c, loopbackmask-2 226a, loopbacknet-1 225c, maskip() 22d, and readipifc() 73c.

6.2.3 /net/ipifc/stats

⟨*function* ipifcstats 77a⟩≡ (384b)
```
int
ipifcstats(Proto *ipifc, char *buf, int len)
{
    return ipstats(ipifc->f, buf, len);
}
```
Uses ipstats() 77b.

⟨*function* ipstats 77b⟩≡ (239a)
```
int
ipstats(Fs *f, char *buf, int len)
{
    IP *ip;
    char *p, *e;
    int i;

    ip = f->ip;
    ip->stats[DefaultTTL] = MAXTTL;

    p = buf;
    e = p+len;
    for(i = 0; i < Nipstats; i++)
        p = seprint(p, e, "%s: %llud\n", statnames[i], ip->stats[i]);
    return p - buf;
}
```
Uses DefaultTTL 31a, MAXTTL 81g, Nipstats 31a, and statnames-265 31b.

6.3 Binding ethernet medium

6.3.1 etherbind()

etherbind is called when the medium's bind method is invoked. It opens three conversations on the Ethernet device: one for IPv4 frames (type 0x0800), one for ARP (type 0x0806), and optionally one for IPv6. For each, it opens /net/etherX/clone, writes the packet type filter, and spawns a reader process that loops reading frames and dispatching them to ipiput4 (for IP) or arpinput (for ARP).

⟨*function* etherbind 77c⟩≡ (366c)
```
/*
 *  called to bind an IP ifc to an ethernet device
 *  called with ifc wlock'd
 */
static void
etherbind(Ipifc *ifc, int argc, char **argv)
{
    Chan *mchan4, *cchan4, *achan, *mchan6, *cchan6, *schan;
    char addr[Maxpath]; //char addr[2*KNAMELEN];
    char dir[Maxpath];  //char dir[2*KNAMELEN];
    char *buf;
    int n;
```

```
char *ptr;
Etherrock *er;

if(argc < 2)
    error(Ebadarg);

mchan4 = cchan4 = achan = mchan6 = cchan6 = nil;
buf = nil;
if(waserror()){
    if(mchan4 != nil)
        cclose(mchan4);
    if(cchan4 != nil)
        cclose(cchan4);
    if(achan != nil)
        cclose(achan);
    if(mchan6 != nil)
        cclose(mchan6);
    if(cchan6 != nil)
        cclose(cchan6);
    if(buf != nil)
        free(buf);
    nexterror();
}

/*
 *  open ipv4 conversation
 *
 *  the dial will fail if the type is already open on
 *  this device.
 */
snprint(addr, sizeof(addr), "%s!0x800", argv[2]);   /* ETIP4 */
mchan4 = chandial(addr, nil, dir, &cchan4);

/*
 *  make it non-blocking
 */
devtab[cchan4->type]->write(cchan4, nbmsg, strlen(nbmsg), 0);

/*
 *  get mac address and speed
 */
snprint(addr, sizeof(addr), "%s/stats", argv[2]);
buf = smalloc(512);
schan = namec(addr, Aopen, OREAD, 0);
if(waserror()){
    cclose(schan);
    nexterror();
}
n = devtab[schan->type]->read(schan, buf, 511, 0);
cclose(schan);
poperror();
buf[n] = 0;

ptr = strstr(buf, "addr: ");
if(!ptr)
    error(Eio);
ptr += 6;
parsemac(ifc->mac, ptr, 6);

ptr = strstr(buf, "mbps: ");
```

```
	if(ptr){
		ptr += 6;
		ifc->mbps = atoi(ptr);
	} else
		ifc->mbps = 100;

	/*
	 *  open arp conversation
	 */
	snprint(addr, sizeof(addr), "%s!0x806", argv[2]);   /* ETARP */
	achan = chandial(addr, nil, nil, nil);

	/*
	 *  open ipv6 conversation
	 *
	 *  the dial will fail if the type is already open on
	 *  this device.
	 */
	snprint(addr, sizeof(addr), "%s!0x86DD", argv[2]);  /* ETIP6 */
	mchan6 = chandial(addr, nil, dir, &cchan6);

	/*
	 *  make it non-blocking
	 */
	devtab[cchan6->type]->write(cchan6, nbmsg, strlen(nbmsg), 0);

	er = smalloc(sizeof(*er));
	er->mchan4 = mchan4;
	er->cchan4 = cchan4;
	er->achan = achan;
	er->mchan6 = mchan6;
	er->cchan6 = cchan6;
	er->f = ifc->conv->p->f;
	ifc->arg = er;

	free(buf);
	poperror();

	kproc("etherread4", etherread4, ifc);
	kproc("recvarpproc", recvarpproc, ifc);
	kproc("etherread6", etherread6, ifc);
}
```

Uses `Maxpath` 232b, `chandial()` 353c, `etherread4()` 359, `etherread6()` 506d, `nbmsg-184` 357d, `parsemac()` 372f, and `recvarpproc()` 364.

6.3.2 chandial()

6.4 Setting up routes

6.5 Advanced configurations

6.5.1 IP Routing

⟨`ipifcctl()` *else if other string* 79⟩≡ (71a) 80c ▷

```
else if(strcmp(argv[0], "iprouting") == 0){
	i = 1;
	if(argc > 1)
```

```
        i = atoi(argv[1]);
    iprouting(cv->p->f, i);
    return nil;
}
```

Uses iprouting() 80a.

⟨*function* iprouting 80a⟩≡ (239a)

```
void
iprouting(Fs *f, bool on)
{
    f->ip->iprouting = on;
    if(f->ip->iprouting == false)
        f->ip->stats[Forwarding] = 2;
    else
        f->ip->stats[Forwarding] = 1;
}
```

Uses Forwarding 31a.

6.5.2 MTU

The MTU (Maximum Transmission Unit) can be set per-interface as a fifth argument to "add". It must fall within the medium's min/max bounds. The MTU determines the maximum IP packet size before fragmentation is required.

⟨ipifcadd() *switch argc, mtu setting case, and fall through* 80b⟩≡ (71c)

```
// add <ip> <mask> <rem> <mtu>
case 5:
    mtu = strtoul(argv[4], 0, 0);
    if(mtu >= ifc->m->mintu && mtu <= ifc->m->maxtu)
        ifc->maxtu = mtu;
    /* fall through */
```

⟨ipifcctl() *else if other string* 80c⟩+≡ (71a) ◁79 82a▷

```
else if(strcmp(argv[0], "mtu") == 0)
    return ipifcsetmtu(ifc, argv, argc);
```

Uses ipifcsetmtu() 80d.

⟨*function* ipifcsetmtu 80d⟩≡ (384b)

```
/*
 *  change an interface's mtu
 */
char*
ipifcsetmtu(Ipifc *ifc, char **argv, int argc)
{
    int mtu;

    if(argc < 2 || ifc->m == nil)
        return Ebadarg;
    mtu = strtoul(argv[1], 0, 0);
    if(mtu < ifc->m->mintu || mtu > ifc->m->maxtu)
        return Ebadarg;
    ifc->maxtu = mtu;
    return nil;
}
```

6.5.3 TOS

TOS (Type of Service) is a byte in the IP header that hints at packet priority. Writing `tos N` to a conversation's `ctl` file sets the value used in outgoing packets.

⟨`Conv(kernel)` *other fields* 81a⟩+≡ (33e) ◁58d 81d▷
```
uint  tos;      /* type of service */
```

⟨`ipwrite()` *Qctl case, else if other string* 81b⟩+≡ (55a) ◁61a 81e▷
```
else if(strcmp(cb->f[0], "tos") == 0)
    tosctlmsg(cv, cb);
```
Uses `tosctlmsg()` 81c.

⟨*function* `tosctlmsg` 81c⟩≡ (331c)
```
static void
tosctlmsg(Conv *c, Cmdbuf *cb)
{
    if(cb->nf < 2)
        c->tos = 0;
    else
        c->tos = atoi(cb->f[1]);
}
```

6.5.4 TTL

TTL (Time To Live) limits how many gateways a packet can traverse—each gateway decrements it by one, and the packet is discarded when TTL reaches zero. This prevents routing loops from circulating packets forever. The default is `MAXTTL`=255.

⟨`Conv(kernel)` *other fields* 81d⟩+≡ (33e) ◁81a 104b▷
```
uint  ttl;      /* max time to live */
```

⟨`ipwrite()` *Qctl case, else if other string* 81e⟩+≡ (55a) ◁81b 104c▷
```
else if(strcmp(cb->f[0], "ttl") == 0)
    ttlctlmsg(cv, cb);
```
Uses `ttlctlmsg()` 81f.

⟨*function* `ttlctlmsg` 81f⟩≡ (331c)
```
static void
ttlctlmsg(Conv *c, Cmdbuf *cb)
{
    if(cb->nf < 2)
        c->ttl = MAXTTL;
    else
        c->ttl = atoi(cb->f[1]);
}
```
Uses `MAXTTL` 81g.

⟨*constant* `MAXTTL` 81g⟩≡ (232b)
```
MAXTTL=   255,
```

6.5.5 Dynamic interface, removing, unbinding, etc.

The ipifc control file also accepts "remove" (to delete an IP address from an interface), "unbind" (to detach the medium entirely), "reassemble" (to enable fragment reassembly on forwarded packets), and several parameters for IPv6 router advertisement. The "remove" command is the inverse of "add"—it deletes the Iplifc node and removes the associated routes.

⟨ipifcctl() *else if other string* 82a⟩+≡ (71a) ◁80c 83a▷

```
else if(strcmp(argv[0], "remove") == 0)
    return ipifcrem(ifc, argv, argc);
```

Uses ipifcrem() 82b.

⟨*function* ipifcrem 82b⟩≡ (384b)

```
/*
 *  remove an address from an interface.
 *  called with c->car locked
 */
char*
ipifcrem(Ipifc *ifc, char **argv, int argc)
{
    char *rv;
    ipaddr ip, mask, rem;
    Iplifc *lifc;

    if(argc < 3)
        return Ebadarg;

    if (parseip(ip, argv[1]) == -1)
        return Ebadip;
    parseipmask(mask, argv[2]);
    if(argc < 4)
        maskip(ip, mask, rem);
    else
        if (parseip(rem, argv[3]) == -1)
            return Ebadip;

    wlock(ifc);

    /*
     *  find address on this interface and remove from chain.
     *  for pt to pt we actually specify the remote address as the
     *  addresss to remove.
     */
    for(lifc = ifc->lifc; lifc != nil; lifc = lifc->next) {
        if (memcmp(ip, lifc->local, IPaddrlen) == 0
        && memcmp(mask, lifc->mask, IPaddrlen) == 0
        && memcmp(rem, lifc->remote, IPaddrlen) == 0)
            break;
    }

    rv = ipifcremlifc(ifc, lifc);
    wunlock(ifc);
    return rv;
}
```

Uses IPaddrlen 20c, maskip() 22d, parseip() 89, and parseipmask() 224d.

⟨*function* ipifcremlifc 82c⟩≡ (384b)

```
/*
 *  remove a logical interface from an ifc
 *  always called with ifc wlock'd
```

```
 */
static char*
ipifcremlifc(Ipifc *ifc, Iplifc *lifc)
{
    Iplifc **l;
    Fs *f;

    f = ifc->conv->p->f;

    /*
     *  find address on this interface and remove from chain.
     *  for pt to pt we actually specify the remote address as the
     *  addresss to remove.
     */
    for(l = &ifc->lifc; *l != nil && *l != lifc; l = &(*l)->next)
        ;
    if(*l == nil)
        return "address not on this interface";
    *l = lifc->next;

    /* disassociate any addresses */
    while(lifc->link)
        remselfcache(f, ifc, lifc, lifc->link->self->a);

    /* remove the route for this logical interface */
    if(isv4(lifc->local))
        v4delroute(f, lifc->remote+IPv4off, lifc->mask+IPv4off, 1);
    ⟨ipifcremlifc() if ipv6 local 518b⟩

    free(lifc);
    return nil;
}
```

⟨ipifcctl() *else if other string* 83a⟩+≡ (71a) ◁82a 105c▷

```
else if(strcmp(argv[0], "unbind") == 0)
    return ipifcunbind(ifc);
```

Uses ipifcunbind() 83b.

⟨*function* ipifcunbind 83b⟩≡ (384b)

```
/*
 *  detach a device from an interface, close the interface
 *  called with ifc->conv closed
 */
static char*
ipifcunbind(Ipifc *ifc)
{
    char *err;

    if(waserror()){
        wunlock(ifc);
        nexterror();
    }
    wlock(ifc);

    /* dissociate routes */
    if(ifc->m != nil && ifc->m->unbindonclose == false)
        ifc->conv->inuse--;
    ifc->ifcid++;

    /* disassociate logical interfaces (before zeroing ifc->arg) */
```

```
    while(ifc->lifc){
        err = ipifcremlifc(ifc, ifc->lifc);
        /*
         * note: err non-zero means lifc not found,
         * which can't happen in this case.
         */
        if(err)
            error(err);
    }

    /* disassociate device */
    if(ifc->m && ifc->m->unbind)
        (*ifc->m->unbind)(ifc);
    memset(ifc->dev, 0, sizeof(ifc->dev));
    ifc->arg = nil;
    ifc->reassemble = false;

    /* close queues to stop queuing of packets */
    qclose(ifc->conv->rq);
    qclose(ifc->conv->wq);
    qclose(ifc->conv->sq);

    ifc->m = nil;
    wunlock(ifc);
    poperror();
    return nil;
}
```

6.5.6 Unbind on close

⟨Medium(kernel) *other fields* 84a⟩≡ (26b)

```
bool unbindonclose;  /* if non-zero, unbind on last close */
```

⟨*function* ipifcclose 84b⟩≡ (384b)

```
/*
 *  called after last close of ipifc data or ctl
 *  called with c locked, we must unlock
 */
static void
ipifcclose(Conv *c)
{
    Ipifc *ifc;
    Medium *m;

    ifc = (Ipifc*)c->ptcl;
    m = ifc->m;
    if(m && m->unbindonclose)
        ipifcunbind(ifc);
}
```

Uses ipifcunbind() 83b.

Chapter 7

Physical Transport: Ethernet

Ethernet is the physical transport layer—the actual wire (or radio) that carries bits between machines on the same local network. Each Ethernet device has a 48-bit MAC address (6 bytes), and frames have a simple header: destination MAC, source MAC, and a 2-byte type field that identifies the payload (0x0800 for IPv4, 0x0806 for ARP).

7.1 Ethernet addresses

Ethernet addresses are 6-byte (48-bit) values, usually written as colon-separated hexadecimal (e.g., `f8:ed:a5:74:92:a` `parseether` converts a string to 6 bytes; `myetheraddr` reads the machine's MAC address from `/net/etherX/addr`.

7.1.1 Parsing

⟨*function* `parseether` 85a⟩≡ (224a)

```
errorneg1
parseether(uchar* to, char *from)
{
    char nip[4];
    char *p;
    int i;

    p = from;
    for(i = 0; i < 6; i++){
        if(*p == '\0')
            return ERROR_NEG1;
        nip[0] = *p++;
        if(*p == '\0')
            return ERROR_NEG1;
        nip[1] = *p++;
        nip[2] = '\0';
        to[i] = strtoul(nip, 0, 16);
        if(*p == ':')
            p++;
    }
    return OK_0;
}
```

7.1.2 `/net/etherx/addr`

⟨*function* `myetheraddr` 85b⟩≡ (225b)

```
errorneg1
myetheraddr(uchar *to, char *dev)
```

```
{
    int n, fd;
    char buf[256];

    if(*dev == '/')
        sprint(buf, "%s/addr", dev);
    else
        sprint(buf, "/net/%s/addr", dev);

    fd = open(buf, OREAD);
    if(fd < 0)
        return ERROR_NEG1;

    n = read(fd, buf, sizeof buf -1 );
    close(fd);
    if(n <= 0)
        return ERROR_NEG1;
    buf[n] = '\0';

    parseether(to, buf);
    return OK_0;
}
```

7.2 Ethernet header

An Ethernet frame header is 14 bytes: destination MAC (`d`), source MAC (`s`), and a 2-byte type field (`t`) identifying the payload protocol (0x0800 for IPv4, 0x0806 for ARP).

⟨*struct* `Etherhdr` 86a⟩≡ (366c)

```
struct Etherhdr
{
    eaddr   d;
    eaddr   s;
    uchar   t[2];
};
```

7.3 Ethernet packet

`Etherpkt` is the full Ethernet frame layout: 6-byte destination, 6-byte source, 2-byte type, then up to 1500 bytes of payload data. The maximum frame size is 1514 bytes (14 header + 1500 payload), which is the standard Ethernet MTU.

⟨*struct* `Etherpkt` 86b⟩≡ (241a)

```
struct Etherpkt
{
  eaddr d;
  eaddr s;
  uchar type[2];

  uchar data[1500];
};
```

7.4 IO

7.4.1 Writing

7.4.2 Reading

7.5 Advanced features

Chapter 8

Inter Network Transport: IP

IP (Internet Protocol) provides the logical addressing layer that enables communication across different physical networks. While Ethernet moves frames between machines on the same wire, IP routes packets between arbitrary machines by hopping through gateways. This chapter covers address parsing, packet output (`ipoput4`), packet input (`ipiput4`), and fragment reassembly.

8.1 IP addresses

IP addresses are stored as binary byte arrays but configured and displayed as dotted-decimal strings (e.g., "10.0.2.15") for IPv4 or colon-separated hex for IPv6. `v4parseip` converts a dotted-decimal string to 4 bytes; `parseip` handles both IPv4 and IPv6 with "::" elision. These parsing functions are used throughout the stack whenever addresses are read from configuration files or user commands.

8.1.1 Parsing

⟨*function* `v4parseip` 88⟩≡ (225a)

```
char*
v4parseip(ipv4 to, char *from)
{
    int i;
    char *p;

    p = from;
    for(i = 0; i < IPv4addrlen && *p; i++){
        to[i] = strtoul(p, &p, 0);
        if(*p == '.')
            p++;
    }
    switch(CLASS(to)){
    case 0: /* class A - 1 uchar net */
    case 1:
        if(i == 3){
            to[3] = to[2];
            to[2] = to[1];
            to[1] = 0;
        } else if (i == 2){
            to[3] = to[1];
            to[1] = 0;
        }
        break;
    case 2: /* class B - 2 uchar net */
        if(i == 3){
```

```
            to[3] = to[2];
            to[2] = 0;
        }
        break;
    }
    return p;
}
```

⟨*function* parseip 89⟩≡ (225a)

```
/*
 * ‘from’ may contain an address followed by other characters,
 * at least in /boot, so we permit whitespace (and more) after the address.
 * we do ensure that "delete" cannot be parsed as "de::".
 *
 * some callers don’t check the return value for errors, so
 * set ‘to’ to something distinctive in the case of a parse error.
 */
vlong
parseip(ipaddr to, char *from)
{
    int i, elipsis = 0;
    bool v4 = true;
    ulong x;
    char *p, *op;

    memset(to, 0, IPaddrlen);
    p = from;
    for(i = 0; i < IPaddrlen && ipcharok(*p); i+=2){
        op = p;
        x = strtoul(p, &p, 16);
        if((*p == '.' && i <= IPaddrlen-4) || (*p == '\0' && i == 0)){
            /* ends with v4 */
            p = v4parseip(to+i, op);
            i += 4;
            break;
        }

        /* v6: at most 4 hex digits, followed by colon or delim */
        if(x != (ushort)x || *p != ':' && !delimchar(*p)) {
            memset(to, 0, IPaddrlen);
            return -1;   /* parse error */
        }
        to[i] = x>>8;
        to[i+1] = x;
        if(*p == ':'){
            v4 = false;
            if(*++p == ':'){ /* :: is elided zero short(s) */
                if (elipsis) {
                    memset(to, 0, IPaddrlen);
                    return -1; /* second :: */
                }
                elipsis = i+2;
                p++;
            }
        } else if (p == op)  /* strtoul made no progress? */
            break;
    }
    if (p == from || !delimchar(*p)) {
        memset(to, 0, IPaddrlen);
        return -1;    /* parse error */
```

```
    }
    if(i < IPaddrlen){
        memmove(&to[elipsis+IPaddrlen-i], &to[elipsis], i-elipsis);
        memset(&to[elipsis], 0, IPaddrlen-i);
    }

    if(v4){
        to[10] = to[11] = 0xff;
        return nhgetl(to + IPv4off);
    } else
        return 6;
}
```

Uses `IPaddrlen` 20c, `IPv4off` 21a, `delimchar()` 224c, and `ipcharok()` 224b.

8.1.2 Comparisons

IP address comparison and equality use byte-wise operations since addresses are 16-byte arrays. `equivip4` compares only the 4 IPv4 bytes (ignoring the v6 prefix) and treats the all-zeros address as a wildcard that matches anything—this is used when looking up conversations by address, where an unset address means "any".

⟨*macro* `ipcmp` 90a⟩≡ (222c)

```
#define ipcmp(x, y) memcmp(x, y, IPaddrlen)
```

⟨*macro* `ipmove` 90b⟩≡ (222c)

```
#define ipmove(x, y) memmove(x, y, IPaddrlen)
```

⟨*function* `equivip4` 90c⟩≡ (227a)

```
bool
equivip4(ipv4 a, ipv4 b)
{
    int i;

    for(i = 0; i < 4; i++)
        if(a[i] != b[i])
            return false;
    return true;
}
```

8.2 IP header

8.2.1 Byte ordering

8.2.2 Checksum

The IP checksum is a one's-complement sum of all 16-bit words in the header. It detects bit errors in transit at the cost of a simple computation. The protocol checksum (`ptclcsum`) handles the trickier case of computing the checksum over a chain of `Block`s that may not be contiguous in memory, carefully tracking whether the current block starts at an odd or even byte boundary.

⟨*function* `ptclcsum` 90d⟩≡ (373)

```
ushort
ptclcsum(Block *bp, int offset, int len)
{
    uchar *addr;
    ulong losum, hisum;
    ushort csum;
```

```
    int odd, blocklen, x;

    /* Correct to front of data area */
    while(bp != nil && offset && offset >= BLEN(bp)) {
        offset -= BLEN(bp);
        bp = bp->next;
    }
    if(bp == nil)
        return 0;

    addr = bp->rp + offset;
    blocklen = BLEN(bp) - offset;

    if(bp->next == nil) {
        if(blocklen < len)
            len = blocklen;
        return ~ptclbsum(addr, len) & 0xffff;
    }

    losum = 0;
    hisum = 0;

    odd = 0;
    while(len) {
        x = blocklen;
        if(len < x)
            x = len;

        csum = ptclbsum(addr, x);
        if(odd)
            hisum += csum;
        else
            losum += csum;
        odd = (odd+x) & 1;
        len -= x;

        bp = bp->next;
        if(bp == nil)
            break;
        blocklen = BLEN(bp);
        addr = bp->rp;
    }

    losum += hisum>>8;
    losum += (hisum&0xff)<<8;
    while((csum = losum>>16) != 0)
        losum = csum + (losum & 0xffff);

    return ~losum & 0xffff;
}
```

⟨*function* ipcsum 91⟩≡ (239a)

```
ushort
ipcsum(uchar *addr)
{
    int len;
    ulong sum;

    sum = 0;
    len = (addr[0]&0xf)<<2;
```

```
    while(len > 0) {
        sum += addr[0]<<8 | addr[1] ;
        len -= 2;
        addr += 2;
    }

    sum = (sum & 0xffff) + (sum >> 16);
    sum = (sum & 0xffff) + (sum >> 16);

    return (sum^0xffff);
}
```

8.3 IP Fragments

IP fragmentation handles packets larger than the MTU of a network link. The sender splits the packet into fragments using the `id` field to identify fragments of the same original packet and the `frag` field for offset and "more fragments" flag. The receiver collects fragments in `Fragment4` structures until all pieces arrive, then reassembles them. Fragments are freed after a timeout to prevent memory leaks from lost fragments.

⟨*function* ipfragfree4 92a⟩≡ (239a)

```
/*
 * ipfragfree4 - Free a list of fragments - assume hold fraglock4
 */
void
ipfragfree4(IP *ip, Fragment4 *frag)
{
    Fragment4 *fl, **l;

    if(frag->blist)
        freeblist(frag->blist);

    frag->src = 0;
    frag->id = 0;
    frag->blist = nil;

    l = &ip->flisthead4;
    for(fl = *l; fl; fl = fl->next) {
        if(fl == frag) {
            *l = frag->next;
            break;
        }
        l = &fl->next;
    }

    frag->next = ip->fragfree4;
    ip->fragfree4 = frag;

}
```

⟨*function* ipfragallo4 92b⟩≡ (239a)

```
/*
 * ipfragallo4 - allocate a reassembly queue - assume hold fraglock4
 */
Fragment4 *
ipfragallo4(IP *ip)
{
    Fragment4 *f;
```

```
    while(ip->fragfree4 == nil) {
        /* free last entry on fraglist */
        for(f = ip->flisthead4; f->next; f = f->next)
            ;
        ipfragfree4(ip, f);
    }
    f = ip->fragfree4;
    ip->fragfree4 = f->next;
    f->next = ip->flisthead4;
    ip->flisthead4 = f;
    f->age = NOW + 30000;

    return f;
}
```

Uses NOW 234a and ipfragfree4() 92a.

8.4 IO

8.4.1 Writing: ipoput4()

`ipoput4` is the single exit point for all outgoing IPv4 packets. It fills in the IP header (version, TTL, TOS), looks up the route to determine the outgoing interface and gateway, and either sends the packet directly (if it fits in one frame) or fragments it if it exceeds the MTU. The final `bwrite` call on the medium sends the packet out the physical interface.

⟨*function* ipoput4 93⟩≡ (239a)

```
int
ipoput4(Fs *f, Block *bp, bool gating, int ttl, int tos, Conv *c)
{
    IP *ip;
    Ip4hdr *eh;
    Ipifc *ifc;
    Route *r, *sr;
    ipv4p gate;
    int len, medialen;
    int rv = OK_0;

    ⟨ipoput4() locals 95b⟩

    ip = f->ip;

    /* Fill out the ip header */
    eh = (Ip4hdr*)(bp->rp);

    ip->stats[OutRequests]++;

    /* Number of uchars in data and ip header to write */
    len = blocklen(bp);

    ⟨ipoput4() if gating 103b⟩

    ⟨ipoput4() error if too big packet of length len 97b⟩

    // Finding the route for the destination!
    r = v4lookup(f, eh->dst, c);
    ⟨ipoput4() error if no route r 97c⟩
```

```
    ifc = r->ifc;
    ⟨ipoput4() set gate according to type of route 94c⟩

    if(!gating) {
        eh->vihl = IP_VER4|IP_HLEN4;
        eh->tos = tos;
    }
    eh->ttl = ttl;

    ⟨ipoput4() rlock ifc, goto free if cant 97e⟩
    ⟨ipoput4() error if no medium attached to interface ifc 97d⟩

    /* If we dont need to fragment just send it */
    ⟨ipoput4() if manual fragmentation setting 104a⟩
    else
        medialen = ifc->maxtu - ifc->m->hsize;

    ⟨ipoput4() if no need to fragment, write simply to medium and return 95a⟩
    ⟨ipoput4() else, need to fragment 95c⟩

raise:
    runlock(ifc);
    poperror();
free:
    freeblist(bp);
    return rv;
}
```

Uses IP_HLEN4 94b, IP_VER4 94a, OutRequests 31a, and v4lookup() 112a.

Header constants

⟨*constant* IP_VER4 94a⟩≡ (232b)

```
IP_VER4=  0x40,
```

⟨*constant* IP_HLEN4 94b⟩≡ (232b)

```
IP_HLEN4= 5,    /* v4: Header length in words */
```

Finding the gateway

If the route is a directly-connected interface (Rifc) or a unicast self-address (Runi), the destination MAC is resolved from the packet's destination IP (via ARP). Otherwise, the packet is forwarded to the route's gateway address, whose MAC will be resolved by ARP at the medium level.

⟨ipoput4() *set gate according to type of route* 94c⟩≡ (93)

```
if(r->type & (Rifc|Runi))
    gate = eh->dst;
else
⟨ipoput4() adjust gate and ifc if broadcast or multicast case 178a⟩
else
    gate = r->v4.gate;
```

Uses Rifc 38e and Runi 38e.

Small send

When the packet fits within the medium's MTU (the common case), `ipoput4` fills in the remaining header fields (packet ID, length, fragment flags, checksum) and calls `ifc->m->bwrite` to hand it to the medium for transmission. The IP checksum covers only the header, not the data—transport protocols add their own checksums.

⟨`ipoput4()` *if no need to fragment, write simply to medium and return* 95a⟩≡ (93)

```
if(len <= medialen) {
    if(!gating)
        hnputs(eh->id, incref(&ip->id4));
    hnputs(eh->length, len);
    // no fragment
    if(!gating){
        eh->frag[0] = 0;
        eh->frag[1] = 0;
    }
    eh->cksum[0] = 0;
    eh->cksum[1] = 0;
    hnputs(eh->cksum, ipcsum(&eh->vihl));
    assert(bp->next == nil);

    // Medium dispatch
    ifc->m->bwrite(ifc, bp, V4, gate);

    runlock(ifc);
    poperror();
    return OK_0;
}
```

Uses `V4` 21g and `ipcsum()` 91.

Fragmentation

When the packet exceeds the MTU, IP splits it into fragments. Each fragment gets its own IP header with the same packet ID but different fragment offsets. The "more fragments" flag (`IP_MF`) is set on all but the last piece. Fragment offsets are in units of 8 bytes, so the payload of each fragment (except the last) must be a multiple of 8. The "don't fragment" flag (`IP_DF`) causes an error instead of fragmentation—useful for path MTU discovery.

⟨`ipoput4()` *locals* 95b⟩≡ (93)

```
Ip4hdr *feh;
ulong fragoff;
Block *xp, *nb;
int lid, seglen, chunk, dlen, blklen, offset;
```

⟨`ipoput4()` *else, need to fragment* 95c⟩≡ (93)

```

if(eh->frag[0] & (IP_DF>>8)){
    if (!gating)
      print("%V: DF set\n", eh->dst);
    ip->stats[FragFails]++;
    ip->stats[OutDiscards]++;
    icmpcantfrag(f, bp, medialen);
    netlog(f, Logip, "%V: eh->frag[0] & (IP_DF>>8)\n", eh->dst);
    goto raise;
}

seglen = (medialen - IP4HDR) & ~7;
if(seglen < 8){
    ip->stats[FragFails]++;
    ip->stats[OutDiscards]++;
```

```
        netlog(f, Logip, "%V seglen < 8\n", eh->dst);
        goto raise;
    }

    dlen = len - IP4HDR;
    xp = bp;
    if(gating)
        lid = nhgets(eh->id);
    else
        lid = incref(&ip->id4);

    offset = IP4HDR;
    while(xp != nil && offset && offset >= BLEN(xp)) {
        offset -= BLEN(xp);
        xp = xp->next;
    }
    xp->rp += offset;

    if(gating)
        fragoff = nhgets(eh->frag)<<3;
    else
        fragoff = 0;
    dlen += fragoff;

    for(; fragoff < dlen; fragoff += seglen) {
        nb = allocb(IP4HDR+seglen);
        feh = (Ip4hdr*)(nb->rp);

        memmove(nb->wp, eh, IP4HDR);
        nb->wp += IP4HDR;

        if((fragoff + seglen) >= dlen) {
            seglen = dlen - fragoff;
            hnputs(feh->frag, fragoff>>3);
        }
        else
            hnputs(feh->frag, (fragoff>>3)|IP_MF);

        hnputs(feh->length, seglen + IP4HDR);
        hnputs(feh->id, lid);

        /* Copy up the data area */
        chunk = seglen;
        while(chunk) {
            if(!xp) {
                ip->stats[OutDiscards]++;
                ip->stats[FragFails]++;
                freeblist(nb);
                netlog(f, Logip, "!xp: chunk %d\n", chunk);
                goto raise;
            }
            blklen = chunk;
            if(BLEN(xp) < chunk)
                blklen = BLEN(xp);
            memmove(nb->wp, xp->rp, blklen);
            nb->wp += blklen;
            xp->rp += blklen;
            chunk -= blklen;
            if(xp->rp == xp->wp)
                xp = xp->next;
```

```
      }

      feh->cksum[0] = 0;
      feh->cksum[1] = 0;
      hnputs(feh->cksum, ipcsum(&feh->vihl));

      // Medium dispatch, send this fragment
      ifc->m->bwrite(ifc, nb, V4, gate);

      ip->stats[FragCreates]++;
  }
  ip->stats[FragOKs]++;
```

Uses `FragCreates` 31a, `FragFails` 31a, `FragOKs` 31a, `IP4HDR` 232b, `IP_DF` 232b, `IP_MF` 232b, `Logip` 233d, `OutDiscards` 31a, `V4` 21g, `icmpcantfrag()` 347b, `ipcsum()` 91, and `netlog()` 389a.

Error managment

⟨*constant* `IP_MAX` 97a⟩≡ (232b)

```
  IP_MAX=   64*1024,  /* Max. Internet packet size, v4 & v6 */
```

⟨`ipoput4()` *error if too big packet of length len* 97b⟩≡ (93)

```
  if(len >= IP_MAX){
      ip->stats[OutDiscards]++;
      netlog(f, Logip, "exceeded ip max size %V\n", eh->dst);
      goto free;
  }
```

Uses `IP_MAX` 97a, `Logip` 233d, `OutDiscards` 31a, and `netlog()` 389a.

⟨`ipoput4()` *error if no route r* 97c⟩≡ (93)

```
  if(r == nil){
      ip->stats[OutNoRoutes]++;
      netlog(f, Logip, "no interface %V\n", eh->dst);
      rv = -1;
      goto free;
  }
```

Uses `Logip` 233d, `OutNoRoutes` 31a, and `netlog()` 389a.

⟨`ipoput4()` *error if no medium attached to interface ifc* 97d⟩≡ (93)

```
  if(ifc->m == nil)
      goto raise;
```

⟨`ipoput4()` *rlock ifc, goto free if cant* 97e⟩≡ (93)

```
  if(!canrlock(ifc))
      goto free;
  if(waserror()){
      runlock(ifc);
      nexterror();
  }
```

8.4.2 Reading: `ipiput4()`

`ipiput4` is called when an IP packet arrives from the network. It validates the header checksum, checks whether the packet is addressed to this machine (`ipforme`), reassembles fragments if needed, then dispatches to the

appropriate protocol handler via `Fsrcvpcol` (which looks up the protocol number in `f->t2p`). If the packet is not for us, it is either forwarded (if this machine is a gateway) or dropped.

⟨*function* `ipiput4` 98⟩≡ (239a)

```
void
ipiput4(Fs *f, Ipifc *ifc, Block *bp)
{
    IP *ip;
    Ip4hdr *h;
    ipaddr v6dst;
    // enum<protocol_type>
    int proto;
    Proto *p;
    bool notforme;
    ⟨ipiput4() locals 99b⟩

    ⟨ipiput4() call ipiput6 if block is not ipv4 515a⟩

    ip = f->ip;
    ip->stats[InReceives]++;

    ⟨ipiput4() ensure we have all the header in the first block 103a⟩

    h = (Ip4hdr*)(bp->rp);

    ⟨ipiput4() check header checksum 99a⟩

    v4tov6(v6dst, h->dst);
    notforme = ipforme(f, v6dst) == 0;

    ⟨ipiput4() check header length and version 99c⟩

    ⟨ipiput4() if notforme 104e⟩
    // else have a ipforme

    ⟨ipiput4() possibly defragment and reassemble 100b⟩

    /* don't let any frag info go up the stack */
    h->frag[0] = 0;
    h->frag[1] = 0;

    proto = h->proto;
    p = Fsrcvpcol(f, proto);

    if(p != nil && p->rcv != nil) {
        ip->stats[InDelivers]++;

        // Protocol dispatch
        (*p->rcv)(p, ifc, bp);
        return;
    }

    ip->stats[InDiscards]++;
    ip->stats[InUnknownProtos]++;
    freeblist(bp);
}
```

Uses `Fsrcvpcol()` 99d, `InDelivers` 31a, `InDiscards` 31a, `InReceives` 31a, `InUnknownProtos` 31a, `ipforme()` 125e, and `v4tov6()` 21c.

Checks

Incoming packets are validated in several steps: the IP header checksum is verified (unless the hardware already checked it via the `Bipck` flag), the header length and version nibbles are checked, and IP options (if present) are stripped from non-forwarded packets. Packets that fail any check are silently dropped and counted as `InHdrErrors`.

⟨ipiput4() *check header checksum* 99a⟩≡ (98)

```
/* dump anything that whose header doesn't checksum */
if((bp->flag & Bipck) == 0 && ipcsum(&h->vihl)) {
    ip->stats[InHdrErrors]++;
    netlog(f, Logip, "ip: checksum error %V\n", h->src);
    freeblist(bp);
    return;
}
```

Uses `InHdrErrors` 31a, `Logip` 233d, `ipcsum()` 91, and `netlog()` 389a.

⟨ipiput4() *locals* 99b⟩≡ (98) 100a ▷

```
int hl;
int olen;
uchar *dp;
```

⟨ipiput4() *check header length and version* 99c⟩≡ (98)

```
/* Check header length and version */
if((h->vihl&0x0F) != IP_HLEN4) {
    hl = (h->vihl&0xF)<<2;
    if(hl < (IP_HLEN4<<2)) {
        ip->stats[InHdrErrors]++;
        netlog(f, Logip, "ip: %V bad hivl %ux\n", h->src, h->vihl);
        freeblist(bp);
        return;
    }
    /* If this is not routed strip off the options */
    if(notforme == false) {
        olen = nhgets(h->length);
        dp = bp->rp + (hl - (IP_HLEN4<<2));
        memmove(dp, h, IP_HLEN4<<2);
        bp->rp = dp;
        h = (Ip4hdr*)(bp->rp);
        h->vihl = (IP_VER4|IP_HLEN4);
        hnputs(h->length, olen-hl+(IP_HLEN4<<2));
    }
}
```

Uses `IP_HLEN4` 94b, `IP_VER4` 94a, `InHdrErrors` 31a, `Logip` 233d, and `netlog()` 389a.

Protocol dispatching

The `proto` byte in the IP header identifies which transport protocol should receive the packet (17 for UDP, 40 for IL, 6 for TCP). `Fsrcvpcol` looks up the protocol in the `t2p` table and calls its `rcv` method, which demultiplexes the packet to the right conversation based on port numbers.

⟨*function* Fsrcvpcol 99d⟩≡ (331c)

```
Proto*
Fsrcvpcol(Fs* f, uchar proto)
{
   return f->t2p[proto];
}
```

Small receive

Reassembling

If the fragment flags are non-zero, the packet is a fragment that must be reassembled before delivery to the transport protocol. `ip4reassemble` collects fragments in a `Fragment4` structure, matching them by source IP, destination IP, and packet ID. When all fragments have arrived (no more "more fragments" flag and the total length is covered), it concatenates the blocks and returns the complete packet. If fragments are still missing, it returns nil and the caller just returns—the packet will be delivered when the last fragment arrives.

⟨ipiput4() *locals* 100a⟩+≡ (98) ◁99b 104d▷

```
  ushort frag;
```

⟨ipiput4() *possibly defragment and reassemble* 100b⟩≡ (98)

```
  frag = nhgets(h->frag);
  if(frag) {
      h->tos = 0;
      if(frag & IP_MF)
          h->tos = 1;
      bp = ip4reassemble(ip, frag, bp, h);
      if(bp == nil)
          return;
      h = (Ip4hdr*)(bp->rp);
  }
```

Uses IP_MF 232b and ip4reassemble() 100c.

⟨*function* ip4reassemble 100c⟩≡ (239a)

```
  Block*
  ip4reassemble(IP *ip, int offset, Block *bp, Ip4hdr *ih)
  {
      int fend;
      ushort id;
      Fragment4 *f, *fnext;
      iplong src, dst;
      Block *bl, **l, *last, *prev;
      int ovlap, len, fragsize, pktposn;

      src = nhgetl(ih->src);
      dst = nhgetl(ih->dst);
      id = nhgets(ih->id);

      /*
       *  block lists are too hard, pullupblock into a single block
       */
      if(bp->next){
          bp = pullupblock(bp, blocklen(bp));
          ih = (Ip4hdr*)(bp->rp);
      }

      qlock(&ip->fraglock4);

      /*
       *  find a reassembly queue for this fragment
       */
      for(f = ip->flisthead4; f; f = fnext){
          fnext = f->next;    /* because ipfragfree4 changes the list */
          if(f->src == src && f->dst == dst && f->id == id)
              break;
          if(f->age < NOW){
              ip->stats[ReasmTimeout]++;
```

```
        ipfragfree4(ip, f);
    }
}

/*
 *  if this isn't a fragmented packet, accept it
 *  and get rid of any fragments that might go
 *  with it.
 */
if(!ih->tos && (offset & ~(IP_MF|IP_DF)) == 0) {
    if(f != nil) {
        ipfragfree4(ip, f);
        ip->stats[ReasmFails]++;
    }
    qunlock(&ip->fraglock4);
    return bp;
}

if(bp->base+IPFRAGSZ >= bp->rp){
    bp = padblock(bp, IPFRAGSZ);
    bp->rp += IPFRAGSZ;
}

BKFG(bp)->foff = offset<<3;
BKFG(bp)->flen = nhgets(ih->length)-IP4HDR;

/* First fragment allocates a reassembly queue */
if(f == nil) {
    f = ipfragallo4(ip);
    f->id = id;
    f->src = src;
    f->dst = dst;

    f->blist = bp;

    qunlock(&ip->fraglock4);
    ip->stats[ReasmReqds]++;
    return nil;
}

/*
 *  find the new fragment's position in the queue
 */
prev = nil;
l = &f->blist;
bl = f->blist;
while(bl != nil && BKFG(bp)->foff > BKFG(bl)->foff) {
    prev = bl;
    l = &bl->next;
    bl = bl->next;
}

/* Check overlap of a previous fragment - trim away as necessary */
if(prev) {
    ovlap = BKFG(prev)->foff + BKFG(prev)->flen - BKFG(bp)->foff;
    if(ovlap > 0) {
        if(ovlap >= BKFG(bp)->flen) {
            freeblist(bp);
            qunlock(&ip->fraglock4);
            return nil;
```

```
		}
		BKFG(prev)->flen -= ovlap;
	}
}

/* Link onto assembly queue */
bp->next = *l;
*l = bp;

/* Check to see if succeeding segments overlap */
if(bp->next) {
	l = &bp->next;
	fend = BKFG(bp)->foff + BKFG(bp)->flen;
	/* Take completely covered segments out */
	while(*l) {
		ovlap = fend - BKFG(*l)->foff;
		if(ovlap <= 0)
			break;
		if(ovlap < BKFG(*l)->flen) {
			BKFG(*l)->flen -= ovlap;
			BKFG(*l)->foff += ovlap;
			/* move up ih hdrs */
			memmove((*l)->rp + ovlap, (*l)->rp, IP4HDR);
			(*l)->rp += ovlap;
			break;
		}
		last = (*l)->next;
		(*l)->next = nil;
		freeblist(*l);
		*l = last;
	}
}

/*
 *  look for a complete packet.  if we get to a fragment
 *  without IP_MF set, we're done.
 */
pktposn = 0;
for(bl = f->blist; bl; bl = bl->next) {
	if(BKFG(bl)->foff != pktposn)
		break;
	if((BLKIP(bl)->frag[0]&(IP_MF>>8)) == 0) {
		bl = f->blist;
		len = nhgets(BLKIP(bl)->length);
		bl->wp = bl->rp + len;

		/* Pullup all the fragment headers and
		 * return a complete packet
		 */
		for(bl = bl->next; bl; bl = bl->next) {
			fragsize = BKFG(bl)->flen;
			len += fragsize;
			bl->rp += IP4HDR;
			bl->wp = bl->rp + fragsize;
		}

		bl = f->blist;
		f->blist = nil;
		ipfragfree4(ip, f);
		ih = BLKIP(bl);
```

```
            hnputs(ih->length, len);
            qunlock(&ip->fraglock4);
            ip->stats[ReasmOKs]++;
            return bl;
        }
        pktposn += BKFG(bl)->flen;
    }
    qunlock(&ip->fraglock4);
    return nil;
}
```

Uses BKFG-267 238, BLKIP-266 39d, IP4HDR 232b, IPFRAGSZ 232d, IP_DF 232b, IP_MF 232b, NOW 234a, ReasmFails 31a, ReasmOKs 31a, ReasmReqds 31a, ReasmTimeout 31a, ipfragallo4() 92b, and ipfragfree4() 92a.

Error managment

Misc

⟨ipiput4() *ensure we have all the header in the first block* 103a⟩≡ (98)

```
/*
 *  Ensure we have all the header info in the first
 *  block.  Make life easier for other protocols by
 *  collecting up to the first 64 bytes in the first block.
 */
if(BLEN(bp) < 64) {
    hl = blocklen(bp);
    if(hl < IP4HDR)
        hl = IP4HDR;
    if(hl > 64)
        hl = 64;
    bp = pullupblock(bp, hl);
    if(bp == nil)
        return;
}
```

Uses IP4HDR 232b.

8.5 Advanced features

When a machine acts as a gateway, packets arrive on one interface and leave on another. The “gating” flag tells ipoput4 that the IP header is already filled in by the original sender—do not overwrite the version, TOS, or ID fields. The gateway only decrements TTL and recomputes the checksum.

8.5.1 Gating

⟨ipoput4() *if gating* 103b⟩≡ (93)

```
if(gating){
    chunk = nhgets(eh->length);
    if(chunk > len){
        ip->stats[OutDiscards]++;
        netlog(f, Logip, "short gated packet\n");
        goto free;
    }
    if(chunk < len)
        len = chunk;
}
```

Uses Logip 233d, OutDiscards 31a, and netlog() 389a.

8.5.2 Manual fragmentation setting

⟨ipoput4() *if manual fragmentation setting* 104a⟩≡ (93)
```
  if(c && c->maxfragsize && c->maxfragsize < ifc->maxtu)
      medialen = c->maxfragsize - ifc->m->hsize;
```

⟨Conv(kernel) *other fields* 104b⟩+≡ (33e) ◁81d
```
  // option<int>, None = 0
  int maxfragsize;    /* If set, used for fragmentation */
```

⟨ipwrite() *Qctl case, else if other string* 104c⟩+≡ (55a) ◁81e 148a▷
```
  else if(strcmp(cb->f[0], "maxfragsize") == 0){
      if(cb->nf < 2)
          error("maxfragsize needs size");

      cv->maxfragsize = (int)strtol(cb->f[1], nil, 0);

  }
```

8.5.3 Routing

When a packet arrives that is not addressed to this machine (`notforme`) and IP routing is enabled, the stack forwards it like a gateway. It decrements TTL, looks up the route for the destination, verifies it doesn't loop back to the source network, optionally reassembles fragments, and sends the packet out the appropriate interface. This is the core of gateway functionality.

⟨ipiput4() *locals* 104d⟩+≡ (98) ◁100a
```
  int hop, tos;
  Route *r;
  Conv conv;
```

⟨ipiput4() *if notforme* 104e⟩≡ (98)
```
      /* route */
      if(notforme) {
          if(!ip->iprouting){
              freeblist(bp);
              return;
          }

          /* don't forward to source's network */
          memset(&conv, 0, sizeof conv);
          conv.r = nil;
          r = v4lookup(f, h->dst, &conv);
          if(r == nil || r->ifc == ifc){
              ip->stats[OutDiscards]++;
              freeblist(bp);
              return;
          }

          /* don't forward if packet has timed out */
          hop = h->ttl;
          if(hop < 1) {
              ip->stats[InHdrErrors]++;
              icmpttlexceeded(f, ifc->lifc->local, bp);
              freeblist(bp);
              return;
          }

          /* reassemble if the interface expects it */
```

```
        if(r->ifc == nil) panic("nil route rfc");
        if(r->ifc->reassemble){
            frag = nhgets(h->frag);
            if(frag) {
                h->tos = 0;
                if(frag & IP_MF)
                    h->tos = 1;
                bp = ip4reassemble(ip, frag, bp, h);
                if(bp == nil)
                    return;
                h = (Ip4hdr*)(bp->rp);
            }
        }

        ip->stats[ForwDatagrams]++;
        tos = h->tos;
        hop = h->ttl;

        ipoput4(f, bp, true, hop - 1, tos, &conv);

        return;
    }
```

Uses `ForwDatagrams` 31a, `IP_MF` 232b, `InHdrErrors` 31a, `OutDiscards` 31a, `icmpttlexceeded()` 346a, `ip4reassemble()` 100c, `ipoput4()` 93, and `v4lookup()` 112a.

⟨`IP(kernel)` *routing fields* 105a⟩≡ (30c)

```
bool iprouting;  /* true if we route like a gateway */
```

8.5.4 Reassembling

⟨`Ipifc(kernel)` *other fields* 105b⟩+≡ (25b) ◁70a

```
bool reassemble; /* reassemble IP packets before forwarding */
```

⟨`ipifcctl()` *else if other string* 105c⟩+≡ (71a) ◁83a 180e▷

```
else if(strcmp(argv[0], "reassemble") == 0){
    ifc->reassemble = true;
    return nil;
}
```

Chapter 9

Finding Machines Locally: ARP

ARP (Address Resolution Protocol) answers the question: "I know the IP address of the machine I want to reach on my local network; what is its Ethernet MAC address?" The ARP cache maps IP addresses to MAC addresses, with entries that expire and are refreshed. When a cache miss occurs, ARP broadcasts a request on the local network and queues the outgoing packet in **hold** until the reply arrives.

⟨Fs(kernel) *arp fields* 106a⟩≡ (29d)

```
  Arp *arp;
```

⟨*struct* Arp 106b⟩≡ (341b)

```
  /*
   *  one per Fs
   */
  struct Arp
  {
      Arpent  *rxmt;
      Proc    *rxmitp;    /* neib sol re-transmit proc */

      Rendez  rxmtq;
      Block   *dropf, *dropl;

      Arpent  *hash[NHASH];
      Arpent  cache[NCACHE];

      // Extra
      QLock;
      ⟨Arp extra fields 106c⟩

  };
```

Uses NCACHE-116 333a and NHASH-115 333a.

⟨Arp *extra fields* 106c⟩≡ (106b)

```
  // ref<Fs>, reverse of Fs.arp
  Fs  *f;
```

⟨*struct* Arpent 106d⟩≡ (234b)

```
  struct Arpent
  {
    ipaddr ip;
    uchar mac[MAClen];

    Medium  *type;      /* media type */

    Block*  hold;
    Block*  last;
```

```
    uint  ctime;      /* time entry was created or refreshed */
    uint  utime;      /* time entry was last used */
    uchar state;
    Arpent  *nextrxt;   /* re-transmit chain */
    uint  rtime;      /* time for next retransmission */
    uchar rxtsrem;

    Ipifc *ifc;
    uchar ifcid;      /* must match ifc->id */

    // Extra
    Arpent* hash;
  };
```

⟨Medium(kernel) *address resolution methods* 107a⟩≡ (26c)

```
  /* address resolution */
  void  (*ares)(Fs*, int, uchar*, uchar*, int, int);  /* resolve */
  void  (*areg)(Ipifc*, uchar*);      /* register */
```

9.1 Initialisation

`arpinit` allocates the ARP cache and spawns the retransmission process (`rxmitproc`), which periodically resends unanswered ARP requests. The ARP cache uses a hash table of `Arpent` entries, each containing a MAC address, an IP address, and a list of packets waiting for resolution.

⟨*function* arpinit 107b⟩≡ (341b)

```
  void
  arpinit(Fs *f)
  {
      f->arp = smalloc(sizeof(Arp));
      f->arp->f = f;
      f->arp->rxmt = nil;
      f->arp->dropf = f->arp->dropl = nil;
      kproc("rxmitproc", rxmitproc, f->arp);
  }
```

Uses rxmitproc() 107c.

⟨*function* rxmitproc 107c⟩≡ (341b)

```
  static void
  rxmitproc(void *v)
  {
      Arp *arp = v;
      long wakeupat;

      arp->rxmitp = up;
      //print("arp rxmitproc started\n");
      if(waserror()){
          arp->rxmitp = 0;
          pexit("hangup", 1);
      }
      for(;;){
          wakeupat = rxmitsols(arp);
          if(wakeupat == 0)
              sleep(&arp->rxmtq, rxready, v);
          else if(wakeupat > ReTransTimer/4)
              tsleep(&arp->rxmtq, returnfalse, 0, wakeupat);
      }
  }
```

Uses ReTransTimer 334a, rxmitsols() 340c, and rxready() 341a.

9.2 /net/arp

⟨ip1gen() *switch TYPE qid cases* 108a⟩≡ (46c) 120a ▷
```
case Qarp:
    p = "arp";
    prot = 0664;
    break;
```
Uses Qarp-217 34i.

⟨ipopen() *switch TYPE qid cases* 108b⟩+≡ (49a) ◁ 62g 182a ▷
```
case Qarp:
case Qiproute:
    if(omode != OREAD && !iseve())
        error(Eperm);
    break;
```
Uses Qarp-217 34i and Qiproute-218 34i.

9.2.1 Reading

Reading `/net/arp` returns one line per cache entry, with the medium name, state ("ok" or "wait"), IP address, and MAC address. Each line is exactly `Alinelen` bytes, which allows random-access reads by offset—the read offset is divided by `Alinelen` to skip entries.

⟨ipread() *switch TYPE qid cases* 108c⟩+≡ (51a) ◁ 65e 120b ▷
```
case Qarp:
    return arpread(f->arp, a, offset, n);
```
Uses Qarp-217 34i and arpread() 108d.

⟨*function* arpread 108d⟩≡ (341b)
```
int
arpread(Arp *arp, char *p, ulong offset, int len)
{
    Arpent *a;
    int n;
    char mac[2*MAClen+1];

    if(offset % Alinelen)
        return 0;

    offset = offset/Alinelen;
    len = len/Alinelen;

    n = 0;
    for(a = arp->cache; len > 0 && a < &arp->cache[NCACHE]; a++){
        if(a->state == 0)
            continue;
        if(offset > 0){
            offset--;
            continue;
        }
        len--;
        qlock(arp);
        convmac(mac, &mac[sizeof mac], a->mac, a->type->maclen);
        n += snprint(p+n, Alinelen+1, aformat, a->type->name,
            arpstate[a->state], a->ip, mac);    /* +1 for NUL */
        qunlock(arp);
    }
```

```
    return n;
}
```
Uses Alinelen-120 339, MAClen 232b, NCACHE-116 333a, aformat 340a, arpstate 333b, and convmac() 340b.

9.2.2 Writing

Writing to /net/arp allows manual manipulation of the cache. The "add" command creates a static entry mapping an IP to a MAC address; "flush" clears all entries; "delete" removes a specific entry. When a new entry is added and there are packets queued in hold waiting for this resolution, they are immediately transmitted.

⟨ipwrite() *switch TYPE qid cases* 109a⟩+≡ (51c) ◁61e 120c▷
```
case Qarp:
    return arpwrite(f, a, n);
```
Uses Qarp-217 34i and arpwrite() 109b.

⟨*function* arpwrite 109b⟩≡ (341b)
```
int
arpwrite(Fs *fs, char *s, int len)
{
    int n;
    Route *r;
    Arp *arp;
    Block *bp;
    Arpent *a, *fl, **l;
    Medium *m;
    char *f[4], buf[256];
    ipaddr ip;
    uchar mac[MAClen];

    arp = fs->arp;

    if(len == 0)
        error(Ebadarp);
    if(len >= sizeof(buf))
        len = sizeof(buf)-1;
    strncpy(buf, s, len);
    buf[len] = 0;
    if(len > 0 && buf[len-1] == '\n')
        buf[len-1] = 0;

    n = getfields(buf, f, 4, 1, " ");
    if(strcmp(f[0], "flush") == 0){
        qlock(arp);
        for(a = arp->cache; a < &arp->cache[NCACHE]; a++){
            memset(a->ip, 0, sizeof(a->ip));
            memset(a->mac, 0, sizeof(a->mac));
            a->hash = nil;
            a->state = 0;
            a->utime = 0;
            while(a->hold != nil){
                bp = a->hold->list;
                freeblist(a->hold);
                a->hold = bp;
            }
        }
        memset(arp->hash, 0, sizeof(arp->hash));
        /* clear all pkts on these lists (rxmt, dropf/l) */
        arp->rxmt = nil;
```

```
	arp->dropf = nil;
	arp->dropl = nil;
	qunlock(arp);
} else if(strcmp(f[0], "add") == 0){
	switch(n){
	default:
		error(Ebadarg);
	case 3:
		if (parseip(ip, f[1]) == -1)
			error(Ebadip);
		if(isv4(ip))
			r = v4lookup(fs, ip+IPv4off, nil);
		else
			r = v6lookup(fs, ip, nil);
		if(r == nil)
			error("Destination unreachable");
		m = r->ifc->m;
		n = parsemac(mac, f[2], m->maclen);
		break;
	case 4:
		m = ipfindmedium(f[1]);
		if(m == nil)
			error(Ebadarp);
		if (parseip(ip, f[2]) == -1)
			error(Ebadip);
		n = parsemac(mac, f[3], m->maclen);
		break;
	}

	if(m->ares == nil)
		error(Ebadarp);

	m->ares(fs, V6, ip, mac, n, 0);
} else if(strcmp(f[0], "del") == 0){
	if(n != 2)
		error(Ebadarg);

	if (parseip(ip, f[1]) == -1)
		error(Ebadip);
	qlock(arp);

	l = &arp->hash[haship(ip)];
	for(a = *l; a; a = a->hash){
		if(memcmp(ip, a->ip, sizeof(a->ip)) == 0){
			*l = a->hash;
			break;
		}
		l = &a->hash;
	}

	if(a){
		/* take out of re-transmit chain */
		l = &arp->rxmt;
		for(fl = *l; fl; fl = fl->nextrxt){
			if(fl == a){
				*l = a->nextrxt;
				break;
			}
			l = &fl->nextrxt;
		}
```

```
            a->nextrxt = nil;
            a->hash = nil;
            a->hold = nil;
            a->last = nil;
            a->ifc = nil;
            memset(a->ip, 0, sizeof(a->ip));
            memset(a->mac, 0, sizeof(a->mac));
        }
        qunlock(arp);
    } else
        error(Ebadarp);

    return len;
}
```

Uses `Ebadarp` 333c, `IPv4off` 21a, `MAClen` 232b, `NCACHE-116` 333a, `V6` 21g, `haship-119` 333d, `ipfindmedium()` 70c, `isv4()` 21b, `parseip()` 89, `parsemac()` 372f, `v4lookup()` 112a, and `v6lookup()` 508b.

Chapter 10

Finding Machines Globally: Routes

While ARP handles local delivery, routing handles inter-network delivery: given a destination IP, which gateway should the packet be forwarded to? The routing table is stored as a forest of ternary search trees (left, mid, right), indexed by a hash of the destination address. `v4lookup` searches this tree to find the matching route, which provides both the next-hop gateway address and the outgoing interface. Results are cached per-conversation to avoid repeated lookups.

10.1 v4lookup()

`v4lookup` first checks the conversation's cached route: if the routing table generation hasn't changed, the cached result is still valid. Otherwise, it hashes the destination address into one of 1024 root buckets and walks the ternary tree, comparing the destination against each node's address range. The walk terminates at the most specific matching route (longest prefix match).

⟨*function* `v4lookup` 112a⟩≡ (369c)

```
Route*
v4lookup(Fs *f, ipv4 a, Conv *c)
{
    Route *p, *q;
    ulong la;
    ⟨v4lookup() locals 113c⟩

    ⟨v4lookup() return cached route if still valid route 113b⟩

    la = nhgetl(a);
    q = nil;
    ⟨v4lookup() ternary search for route q for la in route forest 112b⟩
    ⟨v4lookup() make sure route q has an up to date ifc 113d⟩

    if(c != nil){
        c->r = q;
        c->rgen = v4routegeneration;
    }

    return q;
}
```

Uses `v4routegeneration-122` 37d.

⟨`v4lookup()` *ternary search for route q for la in route forest* 112b⟩≡ (112a)

```
for(p=f->v4root[V4H(la)]; p;)
    if(la >= p->v4.address) {
        if(la <= p->v4.endaddress) {
            q = p;
```

```
            p = p->mid;
        } else
            p = p->right;
    } else
        p = p->left;
```
Uses V4H-131 113a.

⟨*macro* V4H 113a⟩≡ (369c)
```
#define V4H(a)  ((a&0x07ffffff)>>(32-Lroot-5))
```

⟨v4lookup() *return cached route if still valid route* 113b⟩≡ (112a)
```
if(c != nil && c->r != nil && c->r->ifc != nil && c->rgen == v4routegeneration)
    return c->r;
```
Uses v4routegeneration-122 37d.

⟨v4lookup() *locals* 113c⟩≡ (112a)
```
ipaddr gate;
Ipifc *ifc;
```

⟨v4lookup() *make sure route q has an up to date ifc* 113d⟩≡ (112a)
```
if(q && (q->ifc == nil || q->ifcid != q->ifc->ifcid)){
    if(q->type & Rifc) {
        hnputl(gate+IPv4off, q->v4.address);
        memmove(gate, v4prefix, IPv4off);
    } else
        v4tov6(gate, q->v4.gate);

    ifc = findipifc(f, gate, q->type);
    if(ifc == nil)
        return nil;
    q->ifc = ifc;
    q->ifcid = ifc->ifcid;
}
```
Uses IPv4off 21a, Rifc 38e, findipifc() 113e, v4prefix 20e, and v4tov6() 21c.

10.2 findipifc()

⟨*function* findipifc 113e⟩≡ (384b)
```
/*
 *  find the ifc on same net as the remote system.  If none,
 *  return nil.
 */
Ipifc*
findipifc(Fs *f, ipaddr remote, int type)
{
    Ipifc *ifc, *x;
    Iplifc *lifc;
    Conv **cp, **e;
    ipaddr gnet, xmask;

    x = nil;
    memset(xmask, 0, IPaddrlen);

    /* find most specific match */
    e = &f->ipifc->conv[f->ipifc->nc];
    for(cp = f->ipifc->conv; cp < e; cp++){
        if(*cp == nil)
            continue;
```

```
        ifc = (Ipifc*)(*cp)->ptcl;
        for(lifc = ifc->lifc; lifc; lifc = lifc->next){
            maskip(remote, lifc->mask, gnet);
            if(ipcmp(gnet, lifc->net) == 0){
                if(x == nil || ipcmp(lifc->mask, xmask) > 0){
                    x = ifc;
                    ipmove(xmask, lifc->mask);
                }
            }
        }
    }
    if(x != nil)
        return x;

    ⟨findipifc() if broadcast or multicast route 178b⟩
    return nil;
}
```

Uses `IPaddrlen` 20c, `ipcmp` 23d, `ipmove` 23c, and `maskip()` 22d.

10.3 Adding routes: `v4addroute()`

`v4addroute` inserts a new route into the ternary search tree. Each route covers an address range (`address` to `endaddress`) and points to a gateway. The route is inserted using `routeadd`, which walks the tree to find the correct position and may trigger rebalancing. Adding a route also increments the generation counter, invalidating all cached routes in conversations.

⟨*function* `v4addroute` 114⟩≡ (369c)

```
void
v4addroute(Fs *f, char *tag, ipv4 a, ipv4 mask, ipv4 gate, int type)
{
    Route *p;
    ulong m;
    ulong sa; // start address
    ulong ea; // end address
    int h, eh;

    m = nhgetl(mask);
    sa = nhgetl(a) & m;
    ea = sa | ~m;

    eh = V4H(ea);
    for(h=V4H(sa); h<=eh; h++) {
        p = allocroute(Rv4 | type);
        p->v4.address = sa;
        p->v4.endaddress = ea;
        memmove(p->v4.gate, gate, sizeof(p->v4.gate));
        memmove(p->tag, tag, sizeof(p->tag));

        wlock(&routelock);
        addnode(f, &f->v4root[h], p);
        ⟨v4addroute() if f has a route queue 118b⟩
        wunlock(&routelock);
    }
    v4routegeneration++;

    ipifcaddroute(f, Rv4, a, mask, gate, type);
}
```

Uses Rv4 38e, V4H-131 113a, addnode() 117, allocroute() 115c, ipifcaddroute() 133b, routelock-123 115a, and v4routegeneration-122 37d.

⟨*global* routelock 115a⟩≡ (369c)
```
static RWlock   routelock;
```

10.4 Broadcast routes

10.5 Route managment

Route nodes are managed with a free list: `allocroute` reuses freed nodes from the `v4freelist` chain before falling back to `malloc`. `addnode` inserts a route into the ternary tree, splitting existing nodes when address ranges overlap, and calls `balancetree` to keep the tree balanced. The insertion logic handles four cases: the new range is entirely left, entirely right, a subset (goes to `mid`), or overlapping (requiring a split).

⟨*global* v4freelist 115b⟩≡ (369c)
```
/* these are used for all instances of IP */
static Route*   v4freelist;
```

10.5.1 Allocation

⟨*function* allocroute 115c⟩≡ (369c)
```
static Route*
allocroute(int type)
{
    Route *r;
    int n;
    Route **l;

    if(type & Rv4){
        n = sizeof(RouteTree) + sizeof(V4route);
        l = &v4freelist;
    }
    ⟨allocroute() if ipv6 route 518c⟩

    r = *l;
    if(r != nil){
        *l = r->mid;
    } else {
        r = malloc(n);
        if(r == nil)
            panic("out of routing nodes");
    }
    memset(r, 0, n);

    r->type = type;
    r->ifc = nil;
    r->ref = 1;

    return r;
}
```
Uses Rv4 38e and v4freelist-121 115b.

10.5.2 Free

Freed routes are returned to a free list (`v4freelist`) linked through the `mid` pointer. `allocroute` checks this list before calling `malloc`, avoiding memory allocation overhead in the common case of route churn.

⟨*function* `freeroute` 116a⟩≡ (369c)

```
static void
freeroute(Route *r)
{
    Route **l;

    r->left = nil;
    r->right = nil;
    if(r->type & Rv4)
        l = &v4freelist;
    ⟨freeroute() if ipv6 route 518d⟩
    r->mid = *l;
    *l = r;
}
```

Uses `Rv4` 38e and `v4freelist-121` 115b.

10.5.3 Insertion

`addnode` inserts a route into the ternary tree by comparing address ranges. `rangecompare` classifies the relationship between two routes: `Rpreceeds` (goes left), `Rfollows` (goes right), `Requals` (replace the existing route), `Rcontains` (the new route is broader, so the old one goes to `mid`), or `Rcontained` (the new route is narrower and goes to `mid` of the existing one). When a route is contained within another, it provides a more specific match for a subset of addresses.

⟨*enum* `_anon_ (kernel/network/ip/iproute.c)` 116b⟩≡ (369c)

```
/*
 *  compare 2 v4 or v6 ranges
 */
enum
{
    Rpreceeds,
    Rfollows,
    Requals,
    Rcontains,
    Rcontained,
};
```

⟨*function* `rangecompare` 116c⟩≡ (369c)

```
static int
rangecompare(Route *a, Route *b)
{
    if(a->type & Rv4){
        if(a->v4.endaddress < b->v4.address)
            return Rpreceeds;

        if(a->v4.address > b->v4.endaddress)
            return Rfollows;

        if(a->v4.address <= b->v4.address
        && a->v4.endaddress >= b->v4.endaddress){

            if(a->v4.address == b->v4.address
            && a->v4.endaddress == b->v4.endaddress)
                return Requals;
```

```
            return Rcontains;
        }
        return Rcontained;
    }
    ⟨rangecompare() if ipv6 routes 518e⟩
}
```

Uses `Rcontained-130` 116b, `Rcontains-129` 116b, `Requals-128` 116b, `Rfollows-127` 116b, `Rpreceeds-126` 116b, and `Rv4` 38e.

⟨*function* `addnode` 117⟩≡ (369c)

```
/*
 *  add a new node to the tree
 */
static void
addnode(Fs *f, Route **cur, Route *new)
{
    Route *p;

    p = *cur;
    if(p == 0) {
        *cur = new;
        new->depth = 1;
        return;
    }

    switch(rangecompare(new, p)){
    case Rpreceeds:
        addnode(f, &p->left, new);
        break;
    case Rfollows:
        addnode(f, &p->right, new);
        break;
    case Rcontains:
        /*
         *  if new node is superset
         *  of tree node,
         *  replace tree node and
         *  queue tree node to be
         *  merged into root.
         */
        *cur = new;
        new->depth = 1;
        addqueue(&f->queue, p);
        break;
    case Requals:
        /*
         *  supercede the old entry if the old one isn't
         *  a local interface.
         */
        if((p->type & Rifc) == 0){
            p->type = new->type;
            p->ifcid = -1;
            copygate(p, new);
        } else if(new->type & Rifc)
            p->ref++;
        freeroute(new);
        break;
    case Rcontained:
        addnode(f, &p->mid, new);
        break;
    }
```

```
    balancetree(cur);
}
```

Uses Rcontained-130 116b, Rcontains-129 116b, Requals-128 116b, Rfollows-127 116b, Rifc 38e, Rpreceeds-126 116b, addnode() 117, addqueue() 118e, balancetree() 119b, copygate() 118a, freeroute() 116a, and rangecompare() 116c.

⟨*function* copygate 118a⟩≡ (369c)

```
static void
copygate(Route *old, Route *new)
{
    if(new->type & Rv4)
        memmove(old->v4.gate, new->v4.gate, IPv4addrlen);
    else
        memmove(old->v6.gate, new->v6.gate, IPaddrlen);
}
```

Uses IPaddrlen 20c, IPv4addrlen 20a, and Rv4 38e.

⟨v4addroute() *if f has a route queue* 118b⟩≡ (114)

```
while(p = f->queue) {
    f->queue = p->mid;
    walkadd(f, &f->v4root[h], p->left);
    freeroute(p);
}
```

Uses freeroute() 116a and walkadd() 118d.

⟨Fs(kernel) *routing fields* 118c⟩+≡ (29d) ◁39a

```
Route *queue;     /* used as temp when reinjecting routes */
```

⟨*function* walkadd 118d⟩≡ (369c)

```
/*
 *  walk down a tree adding nodes back in
 */
static void
walkadd(Fs *f, Route **root, Route *p)
{
    Route *l, *r;

    l = p->left;
    r = p->right;
    p->left = 0;
    p->right = 0;
    addnode(f, root, p);
    if(l)
        walkadd(f, root, l);
    if(r)
        walkadd(f, root, r);
}
```

Uses addnode() 117 and walkadd() 118d.

⟨*function* addqueue 118e⟩≡ (369c)

```
static void
addqueue(Route **q, Route *r)
{
    Route *l;

    if(r == nil)
        return;

    l = allocroute(r->type);
```

```
    l->mid = *q;
    *q = l;
    l->left = r;
}
```

Uses allocroute() 115c.

10.5.4 Balancing

The routing tree is balanced with AVL-style rotations. Each node tracks its `depth` (height of the subtree), and after every insertion `balancetree` checks whether the left and right subtrees differ by more than one level. If so, it performs a rotation to restore balance, ensuring that `v4lookup` remains $O(\log n)$.

⟨Routetree *other fields* 119a⟩+≡ (38b) ◁70b

```
uchar depth;
```

⟨*function* balancetree 119b⟩≡ (369c)

```
/*
 *  balance the tree at the current node
 */
static void
balancetree(Route **cur)
{
    Route *p, *l, *r;
    int dl, dr;

    /*
     * if left and right are
     * too out of balance,
     * rotate tree node
     */
    p = *cur;
    dl = 0; if(l = p->left) dl = l->depth;
    dr = 0; if(r = p->right) dr = r->depth;

    if(dl > dr+1) {
        p->left = l->right;
        l->right = p;
        *cur = l;
        calcd(p);
        calcd(l);
    } else
    if(dr > dl+1) {
        p->right = r->left;
        r->left = p;
        *cur = r;
        calcd(p);
        calcd(r);
    } else
        calcd(p);
}
```

Uses calcd() 119c.

⟨*function* calcd 119c⟩≡ (369c)

```
/*
 *  calculate depth
 */
static void
calcd(Route *p)
```

```
{
    Route *q;
    int d;

    if(p) {
        d = 0;
        q = p->left;
        if(q)
            d = q->depth;
        q = p->right;
        if(q && q->depth > d)
            d = q->depth;
        q = p->mid;
        if(q && q->depth > d)
            d = q->depth;
        p->depth = d+1;
    }
}
```

10.6 /net/iproutes

`/net/iproute` exposes the routing table as a text file. Reading it returns one line per route with address, mask, gateway, tag, and interface number. Writing it accepts "add" and "remove" commands to manually configure routes—this is how `ipconfig` sets up the default gateway after DHCP.

⟨`ip1gen()` *switch TYPE qid cases* 120a⟩+≡ (46c) ◁108a 130b▷

```
case Qiproute:
    p = "iproute";
    prot = 0664;
    break;
```

Uses `Qiproute-218` 34i.

⟨`ipread()` *switch TYPE qid cases* 120b⟩+≡ (51a) ◁108c 130c▷

```
case Qiproute:
    return routeread(f, a, offset, n);
```

Uses `Qiproute-218` 34i and `routeread()` 120d.

⟨`ipwrite()` *switch TYPE qid cases* 120c⟩+≡ (51c) ◁109a 182c▷

```
case Qiproute:
    return routewrite(f, ch, a, n);
```

Uses `Qiproute-218` 34i and `routewrite()` 123b.

10.6.1 Reading

`routeread` walks the routing forest in-order using `ipwalkroutes`, which visits every node in the ternary tree (left, current, mid, right). The `sprintroute` callback formats each route as a fixed-width text line with address, mask, gateway, type tag, and interface number. The walk supports random-access reads via offset for compatibility with `read` semantics.

⟨*function* `routeread` 120d⟩≡ (369c)

```
long
routeread(Fs *f, char *p, ulong offset, int n)
{
    Routewalk rw;

    rw.p = p;
```

```
    rw.e = p+n;
    rw.o = -offset;
    rw.walk = sprintroute;

    ipwalkroutes(f, &rw);

    return rw.p - p;
}
```
Uses `ipwalkroutes()` 122b and `sprintroute()` 121b.

⟨*global* `rformat` 121a⟩≡ (369c)
```
static char *rformat = "%-15I %-4M %-15I %4.4s %4.4s %3s\n";
```
Uses `rformat-133` 121a.

⟨*function* `sprintroute` 121b⟩≡ (369c)
```
/*
 *  this code is not in rr to reduce stack size
 */
static void
sprintroute(Route *r, Routewalk *rw)
{
    int nifc, n;
    char t[5], *iname, ifbuf[5];
    ipaddr addr, mask, gate;
    char *p;

    convroute(r, addr, mask, gate, t, &nifc);
    iname = "-";
    if(nifc != -1) {
        iname = ifbuf;
        snprint(ifbuf, sizeof ifbuf, "%d", nifc);
    }
    p = seprint(rw->p, rw->e, rformat, addr, mask, gate, t, r->tag, iname);
    if(rw->o < 0){
        n = p - rw->p;
        if(n > -rw->o){
            memmove(rw->p, rw->p-rw->o, n+rw->o);
            rw->p = p + rw->o;
        }
        rw->o += n;
    } else
        rw->p = p;
}
```
Uses `convroute()` 121c and `rformat-133` 121a.

⟨*function* `convroute` 121c⟩≡ (369c)
```
void
convroute(Route *r, uchar *addr, uchar *mask, uchar *gate, char *t, int *nifc)
{
    int i;

    if(r->type & Rv4){
        memmove(addr, v4prefix, IPv4off);
        hnputl(addr+IPv4off, r->v4.address);
        memset(mask, 0xff, IPv4off);
        hnputl(mask+IPv4off, ~(r->v4.endaddress ^ r->v4.address));
        memmove(gate, v4prefix, IPv4off);
        memmove(gate+IPv4off, r->v4.gate, IPv4addrlen);
    } else {
```

```
        for(i = 0; i < IPllen; i++){
            hnputl(addr + 4*i, r->v6.address[i]);
            hnputl(mask + 4*i, ~(r->v6.endaddress[i] ^ r->v6.address[i]));
        }
        memmove(gate, r->v6.gate, IPaddrlen);
    }

    routetype(r->type, t);

    if(r->ifc)
        *nifc = r->ifc->conv->x;
    else
        *nifc = -1;
}
```

Uses IPaddrlen 20c, IPllen 232b, IPv4addrlen 20a, IPv4off 21a, Rv4 38e, routetype() 122a, and v4prefix 20e.

⟨*function* routetype 122a⟩≡ (369c)

```
void
routetype(int type, char *p)
{
    memset(p, ' ', 4);
    p[4] = 0;
    if(type & Rv4)
        *p++ = '4';
    else
        *p++ = '6';
    if(type & Rifc)
        *p++ = 'i';
    if(type & Runi)
        *p++ = 'u';
    else if(type & Rbcast)
        *p++ = 'b';
    else if(type & Rmulti)
        *p++ = 'm';
    if(type & Rptpt)
        *p = 'p';
}
```

Uses Rbcast 38e, Rifc 38e, Rmulti 38e, Rptpt 38e, Runi 38e, and Rv4 38e.

⟨*function* ipwalkroutes 122b⟩≡ (369c)

```
void
ipwalkroutes(Fs *f, Routewalk *rw)
{
    rlock(&routelock);
    if(rw->e > rw->p) {
        for(rw->h = 0; rw->h < nelem(f->v4root); rw->h++)
            if(rr(f->v4root[rw->h], rw) == 0)
                break;
    }
    if(rw->e > rw->p) {
        for(rw->h = 0; rw->h < nelem(f->v6root); rw->h++)
            if(rr(f->v6root[rw->h], rw) == 0)
                break;
    }
    runlock(&routelock);
}
```

Uses routelock-123 115a and rr() 123a.

⟨*function* `rr` 123a⟩≡ (369c)

```
/*
 *  recurse descending tree, applying the function in Routewalk
 */
static int
rr(Route *r, Routewalk *rw)
{
    int h;

    if(rw->e <= rw->p)
        return 0;
    if(r == nil)
        return 1;

    if(rr(r->left, rw) == 0)
        return 0;

    if(r->type & Rv4)
        h = V4H(r->v4.address);
    ⟨rr() else if ipv6 address 515f⟩

    if(h == rw->h)
        rw->walk(r, rw);

    if(rr(r->mid, rw) == 0)
        return 0;

    return rr(r->right, rw);
}
```

Uses `Rv4` 38e, `V4H-131` 113a, and `rr()` 123a.

10.6.2 Writing

Writing to `/net/iproute` accepts "add" and "remove" commands with address, mask, and gateway arguments. "`add 0.0.0.0 0.0.0.0 10.0.2.2`" sets the default gateway. The "tag" field associates routes with the process that created them, allowing cleanup when the process exits.

⟨*function* `routewrite` 123b⟩≡ (369c)

```
long
routewrite(Fs *f, Chan *c, char *p, int n)
{
    int h, changed;
    char *tag;
    Cmdbuf *cb;
    ipaddr addr, mask, gate;
    IPaux *a, *na;
    Route *q;

    cb = parsecmd(p, n);
    if(waserror()){
        free(cb);
        nexterror();
    }

    if(strcmp(cb->f[0], "flush") == 0){
        tag = cb->f[1];
        for(h = 0; h < nelem(f->v4root); h++)
            for(changed = 1; changed;){
                wlock(&routelock);
```

```
            changed = routeflush(f, f->v4root[h], tag);
            wunlock(&routelock);
        }
    for(h = 0; h < nelem(f->v6root); h++)
        for(changed = 1; changed;){
            wlock(&routelock);
            changed = routeflush(f, f->v6root[h], tag);
            wunlock(&routelock);
        }
} else if(strcmp(cb->f[0], "remove") == 0){
    if(cb->nf < 3)
        error(Ebadarg);
    if (parseip(addr, cb->f[1]) == -1)
        error(Ebadip);
    parseipmask(mask, cb->f[2]);
    if(memcmp(addr, v4prefix, IPv4off) == 0)
        v4delroute(f, addr+IPv4off, mask+IPv4off, 1);
    else
        v6delroute(f, addr, mask, 1);
} else if(strcmp(cb->f[0], "add") == 0){
    if(cb->nf < 4)
        error(Ebadarg);
    if(parseip(addr, cb->f[1]) == -1 ||
        parseip(gate, cb->f[3]) == -1)
        error(Ebadip);
    parseipmask(mask, cb->f[2]);
    tag = "none";
    if(c != nil){
        a = c->aux;
        tag = a->tag;
    }
    if(memcmp(addr, v4prefix, IPv4off) == 0)
        v4addroute(f, tag, addr+IPv4off, mask+IPv4off, gate+IPv4off, 0);
    else
        v6addroute(f, tag, addr, mask, gate, 0);
} else if(strcmp(cb->f[0], "tag") == 0) {
    if(cb->nf < 2)
        error(Ebadarg);

    a = c->aux;
    na = newipaux(a->owner, cb->f[1]);
    c->aux = na;
    free(a);
} else if(strcmp(cb->f[0], "route") == 0) {
    if(cb->nf < 2)
        error(Ebadarg);
    if (parseip(addr, cb->f[1]) == -1)
        error(Ebadip);

    q = iproute(f, addr);
    print("%I: ", addr);
    if(q == nil)
        print("no route\n");
    else
        printroute(q);
}

poperror();
free(cb);
return n;
```

```
}
```

Uses IPv4off 21a, iproute() 125a, newipaux() 41c, parsecmd(), parseip() 89, parseipmask() 224d, printroute() 125b, routeflush() 132a, routelock-123 115a, v4addroute() 114, v4delroute() 131a, v4prefix 20e, v6addroute() 507c, and v6delroute() 508a.

⟨*function* iproute 125a⟩≡ (369c)

```
Route *
iproute(Fs *fs, ipaddr ip)
{
    if(isv4(ip))
        return v4lookup(fs, ip+IPv4off, nil);
    else
        return v6lookup(fs, ip, nil);
}
```

Uses IPv4off 21a, isv4() 21b, v4lookup() 112a, and v6lookup() 508b.

⟨*function* printroute 125b⟩≡ (369c)

```
static void
printroute(Route *r)
{
    int nifc;
    char t[5], *iname, ifbuf[5];
    ipaddr addr, mask, gate;

    convroute(r, addr, mask, gate, t, &nifc);
    iname = "-";
    if(nifc != -1) {
        iname = ifbuf;
        snprint(ifbuf, sizeof ifbuf, "%d", nifc);
    }
    print(rformat, addr, mask, gate, t, r->tag, iname);
}
```

Uses convroute() 121c and rformat-133 121a.

10.7 Self cache

The "self cache" (Ipselftab) tracks which IP addresses belong to this machine. ipforme checks whether a destination address in an incoming packet matches one of the machine's own addresses (unicast, broadcast, or multicast). This is how the IP layer decides whether to deliver a packet locally or forward it as a gateway.

⟨Fs(kernel) *other fields* 125c⟩+≡ (29d) ◁32c

```
Ipselftab *self;
```

⟨Iplifc(kernel) *extra fields* 125d⟩+≡ (25e) ◁26a

```
Iplink  *link;    /* addresses linked to this lifc */
```

⟨*function* ipforme 125e⟩≡ (384b)

```
/*
 *  returns
 *  0       - no match
 *  Runi
 *  Rbcast
 *  Rmcast
 */
int
ipforme(Fs *f, ipaddr addr)
{
```

```
    Ipself *p;

    p = f->self->hash[hashipa(addr)];
    for(; p; p = p->next){
        if(ipcmp(addr, p->a) == 0)
            return p->type;
    }

    /* hack to say accept anything */
    if(f->self->acceptall)
        return Runi;
    return 0;
}
```

Uses `Runi` 38e, `hashipa-103` 128a, and `ipcmp` 23d.

10.7.1 Data structures

`Ipselftab` is a hash table of `Ipself` entries, keyed by IP address. Each entry records the address type (unicast, broadcast, multicast) and is linked to one or more interfaces via `Iplink` nodes. The `acceptall` flag is set when any interface has the zero address (wildcard), in which case `ipforme` returns true for all packets.

⟨*struct* `Ipselftab` 126a⟩≡ (384b)

```
struct Ipselftab
{
    int inited;
    bool acceptall;  /* true if an interface has the null address */

    Ipself  *hash[NHASH];   /* hash chains */

    // Extra
    QLock;

};
```

Uses `NHASH-100` 374.

⟨`ipifcinit()` *modify f* 126b⟩+≡ (67) ◁68c

```
f->self = smalloc(sizeof(Ipselftab));   /* hack for ipforme */
```

⟨*struct* `Ipself` 126c⟩≡ (384b)

```
/*
 *  cache of local addresses (addresses we answer to)
 */
struct Ipself
{
    uchar   type;       /* type of address */
    ipaddr   a;

    ulong   expire;

    //Extra
    int ref;
    Ipself  *hnext;     /* next address in the hash table */
    Iplink  *link;      /* binding twixt Ipself and Ipifc */
    Ipself  *next;      /* free list */
};
```

⟨*struct* Iplink 127a⟩≡ (234b)

```
/* binding twixt Ipself and Iplifc */
struct Iplink
{
  ulong expire;

  Ipself  *self;
  Iplifc  *lifc;

  // Extra
  int ref;
  Iplink  *next;    /* free list */
  Iplink  *selflink;  /* next link for this local address */
  Iplink  *lifclink;  /* next link for this ifc */
};
```

10.7.2 Adding IPs

`addselfcache` is called by `ipifcadd` whenever a new address is bound to an interface. If the address already exists (shared between interfaces), it increments the reference count and adds a new `Iplink` to connect it to the new interface. Otherwise, it allocates a new `Ipself` entry and inserts it into the hash table. A corresponding route is also added so the routing table knows this address is local.

⟨*function* addselfcache 127b⟩≡ (384b)

```
/*
 *  add to self routing cache
 *  called with c->car locked
 */
static void
addselfcache(Fs *f, Ipifc *ifc, Iplifc *lifc, ipaddr a, int type)
{
    Ipself *p;
    Iplink *lp;
    int h;

    qlock(f->self);

    /* see if the address already exists */
    h = hashipa(a);
    for(p = f->self->hash[h]; p; p = p->next)
        if(memcmp(a, p->a, IPaddrlen) == 0)
            break;

    /* allocate a local address and add to hash chain */
    if(p == nil){
        p = smalloc(sizeof(*p));
        ipmove(p->a, a);
        p->type = type;
        p->next = f->self->hash[h];
        f->self->hash[h] = p;

        /* if the null address, accept all packets */
        if(ipcmp(a, v4prefix) == 0 || ipcmp(a, IPnoaddr) == 0)
            f->self->acceptall = true;
    }

    /* look for a link for this lifc */
    for(lp = p->link; lp; lp = lp->selflink)
        if(lp->lifc == lifc)
```

```
            break;

    /* allocate a lifc-to-local link and link to both */
    if(lp == nil){
        lp = smalloc(sizeof(*lp));
        lp->ref = 1;
        lp->lifc = lifc;
        lp->self = p;
        lp->selflink = p->link;
        p->link = lp;
        lp->lifclink = lifc->link;
        lifc->link = lp;

        /* add to routing table */
        if(isv4(a))
            v4addroute(f, tifc, a+IPv4off, IPallbits+IPv4off,
                a+IPv4off, type);
        else
            v6addroute(f, tifc, a, IPallbits, a, type);

        if((type & Rmulti) && ifc->m->addmulti != nil)
            (*ifc->m->addmulti)(ifc, a, lifc->local);
    } else
        lp->ref++;

    qunlock(f->self);
}
```

Uses IPaddrlen 20c, IPallbits 227d, IPnoaddr 23b, IPv4off 21a, Rmulti 38e, hashipa-103 128a, ipcmp 23d, ipmove 23c, isv4() 21b, tifc-104 73a, v4addroute() 114, v4prefix 20e, and v6addroute() 507c.

⟨*macro* hashipa 128a⟩≡ (384b)

```
/* quick hash for ip addresses */
#define hashipa(a) ( ( ((a)[IPaddrlen-2]<<8) | (a)[IPaddrlen-1] )%NHASH )
```

10.7.3 Removing IPs

`remselfcache` removes an address from the self cache when an interface is deconfigured. It finds the `Ipself` entry, removes the `Iplink` for this specific interface, and decrements the reference count. When the reference count reaches zero (no interfaces claim this address anymore), the `Ipself` entry is freed and the corresponding route is removed.

⟨*function* remselfcache 128b⟩≡ (384b)

```
/*
 *  Decrement reference for this address on this link.
 *  Unlink from selftab if this is the last ref.
 *  called with c->car locked
 */
static void
remselfcache(Fs *f, Ipifc *ifc, Iplifc *lifc, ipaddr a)
{
    Ipself *p, **l;
    Iplink *link, **l_self, **l_lifc;

    qlock(f->self);

    /* find the unique selftab entry */
    l = &f->self->hash[hashipa(a)];
    for(p = *l; p; p = *l){
```

```
    if(ipcmp(p->a, a) == 0)
        break;
    l = &p->next;
}

if(p == nil)
    goto out;

/*
 *  walk down links from an ifc looking for one
 *  that matches the selftab entry
 */
l_lifc = &lifc->link;
for(link = *l_lifc; link; link = *l_lifc){
    if(link->self == p)
        break;
    l_lifc = &link->lifclink;
}

if(link == nil)
    goto out;

/*
 *  walk down the links from the selftab looking for
 *  the one we just found
 */
l_self = &p->link;
for(link = *l_self; link; link = *l_self){
    if(link == *l_lifc)
        break;
    l_self = &link->selflink;
}

if(link == nil)
    panic("remselfcache");

if(--(link->ref) != 0)
    goto out;

if((p->type & Rmulti) && ifc->m->remmulti != nil)
    (*ifc->m->remmulti)(ifc, a, lifc->local);

/* ref == 0, remove from both chains and free the link */
*l_lifc = link->lifclink;
*l_self = link->selflink;
iplinkfree(link);

if(p->link != nil)
    goto out;

/* remove from routing table */
if(isv4(a))
    v4delroute(f, a+IPv4off, IPallbits+IPv4off, 1);
else
    v6delroute(f, a, IPallbits, 1);

/* no more links, remove from hash and free */
*l = p->next;
ipselffree(p);
```

```
    /* if IPnoaddr, forget */
    if(ipcmp(a, v4prefix) == 0 || ipcmp(a, IPnoaddr) == 0)
        f->self->acceptall = false;

out:
    qunlock(f->self);
}
```
Uses IPallbits 227d, IPnoaddr 23b, IPv4off 21a, Rmulti 38e, hashipa-103 128a, ipcmp 23d, iplinkfree() 377c, ipselffree() 378a, isv4() 21b, v4delroute() 131a, v4prefix 20e, and v6delroute() 508a.

10.7.4 /net/ipselftab

/net/ipselftab is a read-only file that lists all IP addresses the machine considers its own, along with the interface number and type (unicast, broadcast, multicast). This is useful for debugging address configuration and multicast group membership.

⟨Qid *toplevel extra cases* 130a⟩≡ (34i) 181g ▷
```
Qipselftab,
```

⟨ip1gen() *switch TYPE qid cases* 130b⟩+≡ (46c) ◁ 120a 181h ▷
```
case Qipselftab:
    p = "ipselftab";
    prot = 0444;
    break;
```
Uses Qipselftab-219 130a.

⟨ipread() *switch TYPE qid cases* 130c⟩+≡ (51a) ◁ 120b 182b ▷
```
case Qipselftab:
    return ipselftabread(f, a, offset, n);
```
Uses Qipselftab-219 130a and ipselftabread() 130d.

⟨*function* ipselftabread 130d⟩≡ (384b)
```
long
ipselftabread(Fs *f, char *cp, ulong offset, int n)
{
    int i, m, nifc, off;
    Ipself *p;
    Iplink *link;
    char state[8];

    m = 0;
    off = offset;
    qlock(f->self);
    for(i = 0; i < NHASH && m < n; i++){
        for(p = f->self->hash[i]; p != nil && m < n; p = p->next){
            nifc = 0;
            for(link = p->link; link; link = link->selflink)
                nifc++;
            routetype(p->type, state);
            m += snprint(cp + m, n - m, stformat, p->a, nifc, state);
            if(off > 0){
                off -= m;
                m = 0;
            }
        }
    }
    qunlock(f->self);
    return m;
}
```
Uses NHASH-100 374, routetype() 122a, and stformat-107 378b.

10.8 Advanced features

10.8.1 Removing routes

⟨*function* v4delroute 131a⟩≡ (369c)

```
void
v4delroute(Fs *f, ipv4 a, ipv4 mask, bool dolock)
{
    Route **r, *p;
    Route rt;
    int h, eh;
    ulong m;

    m = nhgetl(mask);
    rt.v4.address = nhgetl(a) & m;
    rt.v4.endaddress = rt.v4.address | ~m;
    rt.type = Rv4;

    eh = V4H(rt.v4.endaddress);
    for(h=V4H(rt.v4.address); h<=eh; h++) {
        if(dolock)
            wlock(&routelock);
        r = looknode(&f->v4root[h], &rt);
        if(r) {
            p = *r;
            if(--(p->ref) == 0){
                *r = 0;
                addqueue(&f->queue, p->left);
                addqueue(&f->queue, p->mid);
                addqueue(&f->queue, p->right);
                freeroute(p);
                while(p = f->queue) {
                    f->queue = p->mid;
                    walkadd(f, &f->v4root[h], p->left);
                    freeroute(p);
                }
            }
        }
        if(dolock)
            wunlock(&routelock);
    }
    v4routegeneration++;

    ipifcremroute(f, Rv4, a, mask);
}
```

Uses Rv4 38e, V4H-131 113a, addqueue() 118e, freeroute() 116a, ipifcremroute() 133c, looknode() 131b, routelock-123 115a, v4routegeneration-122 37d, and walkadd() 118d.

⟨*function* looknode 131b⟩≡ (369c)

```
Route**
looknode(Route **cur, Route *r)
{
    Route *p;

    for(;;){
        p = *cur;
        if(p == 0)
            return 0;

        switch(rangecompare(r, p)){
```

```
        case Rcontains:
            return 0;
        case Rpreceeds:
            cur = &p->left;
            break;
        case Rfollows:
            cur = &p->right;
            break;
        case Rcontained:
            cur = &p->mid;
            break;
        case Requals:
            return cur;
        }
    }
}
```

Uses Rcontained-130 116b, Rcontains-129 116b, Requals-128 116b, Rfollows-127 116b, Rpreceeds-126 116b, and rangecompare() 116c.

10.8.2 Flushing routes

⟨*function* routeflush 132a⟩≡ (369c)

```
/*
 *  recurse until one route is deleted
 *    returns 0 if nothing is deleted, 1 otherwise
 */
int
routeflush(Fs *f, Route *r, char *tag)
{
    if(r == nil)
        return 0;
    if(routeflush(f, r->mid, tag))
        return 1;
    if(routeflush(f, r->left, tag))
        return 1;
    if(routeflush(f, r->right, tag))
        return 1;
    if((r->type & Rifc) == 0){
        if(tag == nil || strncmp(tag, r->tag, sizeof(r->tag)) == 0){
            delroute(f, r, 0);
            return 1;
        }
    }
    return 0;
}
```

Uses Rifc 38e, delroute() 132b, and routeflush() 132a.

⟨*function* delroute 132b⟩≡ (369c)

```
/*
 *  this code is not in routeflush to reduce stack size
 */
void
delroute(Fs *f, Route *r, int dolock)
{
    ipaddr addr, mask, gate;
    char t[5];
    int nifc;
```

```
    convroute(r, addr, mask, gate, t, &nifc);
    if(r->type & Rv4)
        v4delroute(f, addr+IPv4off, mask+IPv4off, dolock);
    else
        v6delroute(f, addr, mask, dolock);
}
```

Uses `IPv4off` 21a, `Rv4` 38e, `convroute()` 121c, `v4delroute()` 131a, and `v6delroute()` 508a.

10.8.3 Router boards

⟨`Medium(kernel)` *router methods* 133a⟩≡ (26c)

```
/* routes for router boards */
void  (*addroute)(Ipifc *ifc, int, uchar*, uchar*, uchar*, int);
void  (*remroute)(Ipifc *ifc, int, uchar*, uchar*);
```

⟨*function* `ipifcaddroute` 133b⟩≡ (384b)

```
/*
 * distribute routes to active interfaces like the
 * TRIP linecards
 */
void
ipifcaddroute(Fs *f, int vers, uchar *addr, uchar *mask, uchar *gate, int type)
{
    Medium *m;
    Conv **cp, **e;
    Ipifc *ifc;

    e = &f->ipifc->conv[f->ipifc->nc];
    for(cp = f->ipifc->conv; cp < e; cp++){
        if(*cp != nil) {
            ifc = (Ipifc*)(*cp)->ptcl;
            m = ifc->m;
            if(m && m->addroute)
                m->addroute(ifc, vers, addr, mask, gate, type);
        }
    }
}
```

⟨*function* `ipifcremroute` 133c⟩≡ (384b)

```
void
ipifcremroute(Fs *f, int vers, uchar *addr, uchar *mask)
{
    Medium *m;
    Conv **cp, **e;
    Ipifc *ifc;

    e = &f->ipifc->conv[f->ipifc->nc];
    for(cp = f->ipifc->conv; cp < e; cp++){
        if(*cp != nil) {
            ifc = (Ipifc*)(*cp)->ptcl;
            m = ifc->m;
            if(m && m->remroute)
                m->remroute(ifc, vers, addr, mask);
        }
    }
}
```

Chapter 11

Basic Communication: UDP

UDP (User Datagram Protocol) adds port numbers on top of IP—that's essentially all it does. There is no connection establishment, no reliability, no ordering guarantee. A UDP packet is an IP packet with an 8-byte header containing source port, destination port, length, and checksum. This simplicity makes UDP the right choice for DNS queries, NTP, streaming, and other applications where speed matters more than reliability.

11.1 Protocol initialisation

`udpinit` registers UDP as protocol 17 with the IP stack. It fills in the `Proto` method table with UDP-specific functions: `udpcreate` and `udpclose` manage conversations, `udpconnect` and `udpannounce` handle "connect" and "announce" commands, `udpiput` dispatches incoming packets to the right conversation, and `udpkick` (set by `udpcreate`) adds the UDP header and sends packets out.

⟨*function* `udpinit` 134⟩≡ (405)

```
void
udpinit(Fs *fs)
{
    Proto *udp;

    udp = smalloc(sizeof(Proto));
    udp->priv = smalloc(sizeof(Udppriv));

    udp->name = "udp";
    udp->create = udpcreate;
    udp->close = udpclose;

    udp->connect = udpconnect;
    udp->announce = udpannounce;
    udp->ctl = udpctl;

    udp->rcv = udpiput;

    udp->state = udpstate;
    udp->stats = udpstats;

    udp->advise = udpadvise;

    udp->ipproto = IP_UDPPROTO;

    udp->nc = Nchans;
    udp->ptclsize = sizeof(Udpcb);

    Fsproto(fs, udp);
```

```
}
```

Uses `Fsproto()` 68a, `IP_UDPPROTO-371` 135a, `Nchans` 135b, `udpadvise()` 404, `udpannounce()` 140a, `udpclose()` 139b, `udpconnect()` 139c, `udpcreate()` 139a, `udpctl()` 148e, `udpiput()` 142, `udpstate()` 147c, and `udpstats()` 147b.

⟨*constant* `IP_UDPPROTO` 135a⟩≡ (403b)

```
IP_UDPPROTO = 17,
```

⟨*constant* `Nchans` 135b⟩≡ (232b)

```
Nchans=   1024,
```

11.2 Protocol data structures

`Udppriv` holds the protocol-wide hash table (`Ipht`) used to dispatch incoming packets to the right conversation by looking up the destination port. `Udpcb` (the per-conversation control block stored in `Conv.ptcl`) holds per-connection state such as the "headers" flag for user-level UDP header mode.

11.2.1 `Udppriv`

⟨*struct* `Udppriv` 135c⟩≡ (405)

```
struct Udppriv
{
    // hash<(ipaddr * port) , ref<Conv>>
    Ipht        ht;

    ⟨Udppriv stat fields 137b⟩
};
```

11.2.2 Addresses hashtable: `Ipht`

`Ipht` is a hash table that maps the 4-tuple (remote IP, remote port, local IP, local port) to a `Conv`. This is how `udpiput` quickly finds the conversation for an incoming packet: it hashes the packet's addresses and ports, then walks the chain. The hash function uses all four values and reduces modulo 521 (a prime). Wildcard matching (port 0 or address 0) is handled by trying progressively less specific lookups.

⟨*struct* `Ipht` 135d⟩≡ (234b)

```
/*
 *  hash table for 2 ip addresses + 2 ports
 */
struct Ipht
{
  // hash<ipconvhash, ref<Conv>>, next = Iphash.next
  Iphash  *tab[Nipht];

  // Extra
  Lock;
};
```

⟨*constant* `Nipht` 135e⟩≡ (233c)

```
Nipht=    521,  /* convenient prime */
```

⟨*struct* Iphash 136a⟩≡ (234b)

```
struct Iphash
{
  Conv  *c;
  // enum<matchtype>
  int match;

  // Extra
  Iphash  *next;
};
```

⟨*enum* matchtype 136b⟩≡ (234b)

```
enum matchtype {
  IPmatchexact= 0,  /* match on 4 tuple */

  IPmatchany,   /* *!* */
  IPmatchport,    /* *!port */
  IPmatchaddr,    /* addr!* */
  IPmatchpa,    /* addr!port */
};
```

⟨*function* iphash 136c⟩≡ (373)

```
/*
 *  hashing tcp, udp, ... connections
 */
ulong
iphash(ipaddr sa, ushort sp, ipaddr da, ushort dp)
{
    return ((sa[IPaddrlen-1]<<24) ^ (sp << 16) ^ (da[IPaddrlen-1]<<8) ^ dp )
      % Nipht; // pad's first network bugfix :)
}
```

Uses IPaddrlen 20c and Nipht 135e.

⟨*function* iphtadd 136d⟩≡ (373)

```
void
iphtadd(Ipht *ht, Conv *c)
{
    ulong hv;
    Iphash *h;

    hv = iphash(c->raddr, c->rport, c->laddr, c->lport);
    h = smalloc(sizeof(Iphash));
    if(ipcmp(c->raddr, IPnoaddr) != 0)
        h->match = IPmatchexact;
    else {
        if(ipcmp(c->laddr, IPnoaddr) != 0){
            if(c->lport == 0)
                h->match = IPmatchaddr;
            else
                h->match = IPmatchpa;
        } else {
            if(c->lport == 0)
                h->match = IPmatchany;
            else
                h->match = IPmatchport;
        }
    }
    h->c = c;

    lock(ht);
```

```
    // add_hash(h, ht)
    h->next = ht->tab[hv];
    ht->tab[hv] = h;
    unlock(ht);
}
```

Uses IPmatchaddr 136b, IPmatchany 136b, IPmatchexact 136b, IPmatchpa 136b, IPmatchport 136b, IPnoaddr 23b, ipcmp 23d, and iphash() 136c.

⟨*function* iphtrem 137a⟩≡ (373)

```
void
iphtrem(Ipht *ht, Conv *c)
{
    ulong hv;
    Iphash **l, *h;

    hv = iphash(c->raddr, c->rport, c->laddr, c->lport);
    lock(ht);
    // del_hash(hv, ht)
    for(l = &ht->tab[hv]; (*l) != nil; l = &(*l)->next)
        if((*l)->c == c){
            h = *l;
            (*l) = h->next;
            free(h);
            break;
        }
    unlock(ht);
}
```

Uses iphash() 136c.

11.2.3 Statistics

⟨Udppriv *stat fields* 137b⟩≡ (135c)

```
/* MIB counters */
Udpstats    ustats;
/* non-MIB stats */
ulong       csumerr;        /* checksum errors */
ulong       lenerr;         /* short packet */
```

⟨*struct* Udpstats 137c⟩≡ (405)

```
/* MIB II counters */
struct Udpstats
{
    uvlong  udpInDatagrams;
    ulong   udpNoPorts;
    ulong   udpInErrors;
    uvlong  udpOutDatagrams;
};
```

11.3 Protocol header

`Udp4hdr` maps directly to the on-the-wire format: the first 20 bytes are the IP header (reusing `Ip4hdr` field names), followed by an 8-byte UDP header with source port, destination port, length, and checksum. The

structure is laid out so that casting a `Block`'s read pointer to `Udp4hdr*` gives direct access to all fields without parsing.

⟨*struct* `Udp4hdr` 138a⟩≡ (405)
```
struct Udp4hdr
{
    /* ip header */
    uchar   vihl;       /* Version and header length */
    uchar   tos;        /* Type of service */
    uchar   length[2];  /* packet length */
    uchar   id[2];      /* Identification */
    uchar   frag[2];    /* Fragment information */
    uchar   Unused; // ttl

    uchar   udpproto;   /* Protocol */
    uchar   udpplen[2]; /* Header plus data length */
    ipv4    udpsrc;    /* Ip source */
    ipv4    udpdst;    /* Ip destination */

    /* udp header */
    uchar   udpsport[2];    /* Source port */
    uchar   udpdport[2];    /* Destination port */
    uchar   udplen[2];  /* data length */
    uchar   udpcksum[2];    /* Checksum */
};
```

⟨*constant* `UDP4_IPHDR_SZ` 138b⟩≡ (403b)
```
UDP4_IPHDR_SZ = 20,
```

⟨*constant* `UDP_UDPHDR_SZ` 138c⟩≡ (403b)
```
UDP_UDPHDR_SZ   = 8,
```

⟨*constant* `UDP4_PHDR_OFF` 138d⟩≡ (403b)
```
UDP4_PHDR_OFF = 8,
```

⟨*constant* `UDP4_PHDR_SZ` 138e⟩≡ (403b)
```
UDP4_PHDR_SZ = 12,
```

⟨*struct Udphdr (user)* 138f⟩≡ (222c)
```
struct Udphdr
{
    ipaddr raddr; /* V6 remote address */
    ipaddr laddr; /* V6 local address */
    ipaddr ifcaddr; /* V6 ifc addr msg was received on */

    uchar rport[2];  /* remote port */
    uchar lport[2];  /* local port */
};
```

11.4 /net/tcp/clone

`udpcreate` sets up a UDP conversation with a message-oriented read queue (`Qmsg`, so each `read` returns exactly one datagram) and a bypass write queue that calls `udpkick` directly—there is no buffering on the write side

because each **write** produces exactly one UDP packet. **udpclose** removes the conversation from the hash table and resets all addresses.

⟨*function* udpcreate 139a⟩≡ (405)
```
static void
udpcreate(Conv *cv)
{
    cv->rq = qopen(128*1024, Qmsg, 0, 0);
    cv->wq = qbypass(udpkick, cv);
}
```
Uses udpkick() 140b.

⟨*function* udpclose 139b⟩≡ (405)
```
static void
udpclose(Conv *c)
{
    Udpcb *ucb;
    Udppriv *upriv;

    upriv = c->p->priv;
    iphtrem(&upriv->ht, c);

    c->state = Idle;
    qclose(c->rq);
    qclose(c->wq);
    qclose(c->eq);
    ipmove(c->laddr, IPnoaddr);
    ipmove(c->raddr, IPnoaddr);
    c->lport = 0;
    c->rport = 0;

    ucb = (Udpcb*)c->ptcl;
    ucb->headers = 0;
}
```
Uses IPnoaddr 23b, Idle 34b, iphtrem() 137a, and ipmove 23c.

11.5 /net/tcp/x/ctl

UDP's "connect" simply sets the remote address/port (via **Fsstdconnect**), marks the conversation as connected (via **Fsconnected**), and adds it to the hash table. There is no handshake—UDP is connectionless. "Announce" sets the local port and adds the conversation to the hash table so incoming packets to that port will be delivered to it.

11.5.1 Connect

⟨*function* udpconnect 139c⟩≡ (405)
```
static char*
udpconnect(Conv *c, char **argv, int argc)
{
    char *err;
    Udppriv *upriv;

    upriv = c->p->priv;

    err = Fsstdconnect(c, argv, argc);
    Fsconnected(c, err);
    if(err != nil)
```

```
        return err;

    iphtadd(&upriv->ht, c);
    return nil;
}
```

Uses `Fsstdconnect()` 56b and `iphtadd()` 136d.

11.5.2 Announce

⟨*function* `udpannounce` 140a⟩≡ (405)

```
static char*
udpannounce(Conv *c, char** argv, int argc)
{
    char *err;
    Udppriv *upriv;

    upriv = c->p->priv;

    err = Fsstdannounce(c, argv, argc);
    if(err != nil)
        return err;
    Fsconnected(c, nil);

    iphtadd(&upriv->ht, c);

    return nil;
}
```

Uses `Fsstdannounce()` 58b and `iphtadd()` 136d.

11.6 IO

11.6.1 Writing: `udpkick()`

`udpkick` prepends a UDP header (source port, destination port, length, checksum) to the data block, then calls `ipoput4` to send it as an IP packet. The "kick" name comes from the queue model: writing to a conversation's `data` file enqueues a block, and the kick function is called to actually transmit it.

⟨*function* `udpkick` 140b⟩≡ (405)

```
void
udpkick(void *x, Block *bp)
{
    Conv *c = x;
    Conv *rc;
    Udp4hdr *uh4;
    Udppriv *upriv;
    int dlen, ptcllen;
    Fs *f;
    int version;
    ⟨udpkick() locals 148g⟩

    upriv = c->p->priv;
    f = c->p->f;

//  netlog(c->p->f, Logudp, "udp: kick\n"); /* frequent and uninteresting */
    if(bp == nil)
        return;
```

```
    ⟨udpkick() special headers processing 149a⟩
    ⟨udpkick() set version to V4 or V6 514b⟩

    dlen = blocklen(bp);

    /* fill in pseudo header and compute checksum */
    switch(version){
    case V4:
        bp = padblock(bp, UDP4_IPHDR_SZ+UDP_UDPHDR_SZ);
        if(bp == nil)
            return;

        uh4 = (Udp4hdr *)(bp->rp);
        ptcllen = dlen + UDP_UDPHDR_SZ;
        uh4->Unused = 0;
        uh4->udpproto = IP_UDPPROTO;
        uh4->frag[0] = 0;
        uh4->frag[1] = 0;
        hnputs(uh4->udpplen, ptcllen);
        ⟨udpkick() if special headers 149b⟩
        else {
            v6tov4(uh4->udpdst, c->raddr);
            hnputs(uh4->udpdport, c->rport);
            if(ipcmp(c->laddr, IPnoaddr) == 0)
                findlocalip(f, c->laddr, c->raddr);
            v6tov4(uh4->udpsrc, c->laddr);
            rc = c;
        }
        hnputs(uh4->udpsport, c->lport);
        hnputs(uh4->udplen, ptcllen);

        uh4->udpcksum[0] = 0;
        uh4->udpcksum[1] = 0;
        hnputs(uh4->udpcksum,
               ptclcsum(bp, UDP4_PHDR_OFF, dlen+UDP_UDPHDR_SZ+UDP4_PHDR_SZ));
        uh4->vihl = IP_VER4;

        // Let's go, let's send the data
        ipoput4(f, bp, false, c->ttl, c->tos, rc);

        break;

    ⟨udpkick() switch version ipv6 case 514a⟩

    default:
        panic("udpkick: version %d", version);
    }
    upriv->ustats.udpOutDatagrams++;
}
```

Uses `IP_UDPPROTO-371` 135a, `IP_VER4` 94a, `IPnoaddr` 23b, `UDP4_IPHDR_SZ-367` 138b, `UDP4_PHDR_OFF-365` 138d, `UDP4_PHDR_SZ-366` 138e, `UDP_UDPHDR_SZ-364` 138c, `V4` 21g, `findlocalip()` 379, `ipcmp` 23d, `ipoput4()` 93, `ptclcsum()` 90d, and `v6tov4()` 21d.

11.6.2 Reading: `udpiput()`

`udpiput` is called by `ipiput4` when an incoming packet has protocol type UDP. It extracts the source and destination ports from the UDP header, looks up the matching conversation, and enqueues the data on that

conversation's read queue. If no matching conversation is found, the packet is dropped.

⟨*function* udpiput 142⟩≡ (405)
```
void
udpiput(Proto *udp, Ipifc *ifc, Block *bp)
{
    Udppriv *upriv;
    Fs *f;
    Udp4hdr *uh4;
    int version;
    ipaddr raddr, laddr;
    ushort rport, lport;
    Conv *c;
    Udpcb *ucb;
    int len;
    ⟨udpiput() locals 145a⟩
    uchar *p;

    upriv = udp->priv;
    upriv->ustats.udpInDatagrams++;
    f = udp->f;
    uh4 = (Udp4hdr*)(bp->rp);
    ⟨udpiput() set version to V4 or V6 516c⟩

    ⟨udpiput() checking checksum and setting rxxx, lxxx 145b⟩

    qlock(udp);

    // find the corresponding conversation
    c = iphtlook(&upriv->ht, raddr, rport, laddr, lport);
    ⟨udpiput() if no conversation found 146b⟩

    ucb = (Udpcb*)c->ptcl;

    if(c->state == Announced){
        if(ucb->headers == 0){
            /* create a new conversation */
            ⟨udpiput() new conv to create, adjust laddr if not Runi 148b⟩
            c = Fsnewcall(c, raddr, rport, laddr, lport, version);
            if(c == nil){
                qunlock(udp);
                freeblist(bp);
                return;
            }
            // port may have changed?
            iphtadd(&upriv->ht, c);
            ucb = (Udpcb*)c->ptcl;
        }
    }

    qlock(c);
    qunlock(udp);


    ⟨udpiput() trim the packet, adjust bp removing header 146a⟩

    netlog(f, Logudpmsg, "udp: %I.%d -> %I.%d l %d\n", raddr, rport,
           laddr, lport, len);

    ⟨udpiput() if special headers 149c⟩
```

```
    if(bp->next)
        bp = concatblock(bp);

    ⟨udpiput() if reading queue is full 147a⟩

    qpass(c->rq, bp);
    qunlock(c);

}
```
Uses `Announced` 34b, `Fsnewcall()` 144, `Logudpmsg` 233d, `iphtadd()` 136d, `iphtlook()` 143, and `netlog()` 389a.

Conversation dispatch

⟨*function* `iphtlook` 143⟩≡ (373)
```
/* look for a matching conversation with the following precedence
 *  connected && raddr,rport,laddr,lport
 *  announced && laddr,lport
 *  announced && *,lport
 *  announced && laddr,*
 *  announced && *,*
 */
Conv*
iphtlook(Ipht *ht, ipaddr sa, ushort sp, ipaddr da, ushort dp)
{
    ulong hv;
    Iphash *h;
    Conv *c;

    /* exact 4 pair match (connection) */
    hv = iphash(sa, sp, da, dp);
    lock(ht);
    for(h = ht->tab[hv]; h != nil; h = h->next){
        if(h->match == IPmatchexact) {
          c = h->c;
          if(sp == c->rport && dp == c->lport
             && ipcmp(sa, c->raddr) == 0 && ipcmp(da, c->laddr) == 0){
              unlock(ht);
              return c;
          }
        }
    }

    /* match local address and port */
    hv = iphash(IPnoaddr, 0, da, dp);
    for(h = ht->tab[hv]; h != nil; h = h->next){
        if(h->match == IPmatchpa) {
          c = h->c;
          if(dp == c->lport && ipcmp(da, c->laddr) == 0){
              unlock(ht);
              return c;
          }
       }
    }

    /* match just port */
    hv = iphash(IPnoaddr, 0, IPnoaddr, dp);
    for(h = ht->tab[hv]; h != nil; h = h->next){
        if(h->match == IPmatchport) {
```

```
          c = h->c;
          if(dp == c->lport){
              unlock(ht);
              return c;
          }
        }
    }

    /* match local address */
    hv = iphash(IPnoaddr, 0, da, 0);
    for(h = ht->tab[hv]; h != nil; h = h->next){
        if(h->match == IPmatchaddr) {
          c = h->c;
          if(ipcmp(da, c->laddr) == 0){
              unlock(ht);
              return c;
          }
        }
    }

    /* look for something that matches anything */
    hv = iphash(IPnoaddr, 0, IPnoaddr, 0);
    for(h = ht->tab[hv]; h != nil; h = h->next){
        if(h->match == IPmatchany) {
          c = h->c;
          unlock(ht);
          return c;
        }
    }
    unlock(ht);
    return nil;
}
```

Uses `IPmatchaddr` 136b, `IPmatchany` 136b, `IPmatchexact` 136b, `IPmatchpa` 136b, `IPmatchport` 136b, `IPnoaddr` 23b, `ipcmp` 23d, and `iphash()` 136c.

New conversation

⟨*function* `Fsnewcall` 144⟩≡ (331c)

```
/*
 *  called with protocol locked
 */
Conv*
Fsnewcall(Conv *c, ipaddr raddr, ushort rport, ipaddr laddr, ushort lport, uchar version)
{
    Conv *nc;
    Conv **l;
    int i;

    qlock(c);
    i = 0;
    for(l = &c->incall; *l; l = &(*l)->next)
        i++;
    if(i >= Maxincall) {
        static bool beenhere;

        qunlock(c);
        if (!beenhere) {
            beenhere = true;
            print("Fsnewcall: incall queue full (%d) on port %d\n",
```

```
                i, c->lport);
        }
        return nil;
    }

    /* find a free conversation */
    nc = Fsprotoclone(c->p, network);
    if(nc == nil) {
        qunlock(c);
        return nil;
    }
    ipmove(nc->raddr, raddr);
    nc->rport = rport;
    ipmove(nc->laddr, laddr);
    nc->lport = lport;

    nc->next = nil;
    *l = nc;

    nc->state = Connected;
    nc->ipversion = version;

    qunlock(c);

    wakeup(&c->listenr);

    return nc;
}
```

Uses `Connected` 34b, `Fsprotoclone()` 52b, `Maxincall` 232b, `ipmove` 23c, and `network-249` 46a.

Checks

⟨`udpiput()` *locals* 145a⟩≡ (142) 515b ▷

```
int ottl, olen;
```

⟨`udpiput()` *checking checksum and setting rxxx, lxxx* 145b⟩≡ (142)

```
/* Put back pseudo header for checksum
 * (remember old values for icmpnoconv()) */
switch(version) {
case V4:
    ottl = uh4->Unused;
    uh4->Unused = 0;
    len = nhgets(uh4->udplen);
    olen = nhgets(uh4->udpplen);
    hnputs(uh4->udpplen, len);

    v4tov6(raddr, uh4->udpsrc);
    v4tov6(laddr, uh4->udpdst);
    lport = nhgets(uh4->udpdport);
    rport = nhgets(uh4->udpsport);

    if(nhgets(uh4->udpcksum)) {
        if(ptclcsum(bp, UDP4_PHDR_OFF, len+UDP4_PHDR_SZ)) {
            upriv->ustats.udpInErrors++;
            netlog(f, Logudp, "udp: checksum error %I\n", raddr);
            DPRINT("udp: checksum error %I\n", raddr);
            freeblist(bp);
            return;
        }
```

```
        }
        uh4->Unused = ottl;
        hnputs(uh4->udpplen, olen);
        break;

    ⟨udpiput() checking checksum ipv6 case 515c⟩

    default:
        panic("udpiput: version %d", version);
        return; /* to avoid a warning */
    }
```

Uses DPRINT-363 403a, Logudp 233d, UDP4_PHDR_OFF-365 138d, UDP4_PHDR_SZ-366 138e, V4 21g, netlog() 389a, ptclcsum() 90d, and v4tov6() 21c.

Trim the packet

⟨udpiput() *trim the packet, adjust bp removing header* 146a⟩≡ (142)

```
    /*
     * Trim the packet down to data size
     */
    len -= UDP_UDPHDR_SZ;
    switch(version){
    case V4:
        bp = trimblock(bp, UDP4_IPHDR_SZ+UDP_UDPHDR_SZ, len);
        break;
    ⟨udpiput() trim the packet, ipv6 case 515e⟩
    default:
        bp = nil;
        panic("udpiput4: version %d", version);
    }
    if(bp == nil){
        qunlock(c);
        netlog(f, Logudp, "udp: len err %I.%d -> %I.%d\n", raddr, rport,
               laddr, lport);
        upriv->lenerr++;
        return;
    }
```

Uses Logudp 233d, UDP4_IPHDR_SZ-367 138b, UDP_UDPHDR_SZ-364 138c, V4 21g, and netlog() 389a.

Error managment

⟨udpiput() *if no conversation found* 146b⟩≡ (142)

```
    if(c == nil){
        /* no conversation found */
        upriv->ustats.udpNoPorts++;
        qunlock(udp);
        netlog(f, Logudp, "udp: no conv %I!%d -> %I!%d\n", raddr, rport,
               laddr, lport);

        switch(version){
        case V4:
            icmpnoconv(f, bp);
            break;
         ⟨udpiput() no conversation found, ipv6 case 515d⟩
        default:
            panic("udpiput2: version %d", version);
        }
```

```
        freeblist(bp);
        return;
    }
```

Uses `Logudp` 233d, `V4` 21g, `icmpnoconv()` 347a, and `netlog()` 389a.

⟨`udpiput()` *if reading queue is full* 147a⟩≡ (142)

```
    if(qfull(c->rq)){
        qunlock(c);
        netlog(f, Logudp, "udp: qfull %I.%d -> %I.%d\n", raddr, rport,
               laddr, lport);
        freeblist(bp);
        return;
    }
```

Uses `Logudp` 233d and `netlog()` 389a.

11.7 Other files

11.7.1 /net/tcp/stats

⟨*function* `udpstats` 147b⟩≡ (405)

```
    int
    udpstats(Proto *udp, char *buf, int len)
    {
        Udppriv *upriv;

        upriv = udp->priv;
        return snprint(buf, len, "InDatagrams: %llud\nNoPorts: %lud\n"
            "InErrors: %lud\nOutDatagrams: %llud\n",
            upriv->ustats.udpInDatagrams,
            upriv->ustats.udpNoPorts,
            upriv->ustats.udpInErrors,
            upriv->ustats.udpOutDatagrams);
    }
```

11.7.2 /net/tcp/x/status

⟨*function* `udpstate` 147c⟩≡ (405)

```
    static int
    udpstate(Conv *cv, char *state, int n)
    {
        return snprint(state, n, "%s qin %d qout %d\n",
            cv->inuse ? "Open" : "Closed",
            cv->rq ? qlen(cv->rq) : 0,
            cv->wq ? qlen(cv->wq) : 0
        );
    }
```

11.8 Advanced features

11.8.1 Ignore advice

⟨`Conv(kernel)` *udp fields* 147d⟩≡ (33e)

```
    bool ignoreadvice;   /* don't terminate connection on icmp errors */
```

⟨ipwrite() *Qctl case, else if other string* 148a⟩+≡ (55a) ◁104c 180c▷
```
else if(strcmp(cb->f[0], "ignoreadvice") == 0)
    cv->ignoreadvice = true;
```

11.8.2 Not regular forme

⟨udpiput() *new conv to create, adjust laddr if not Runi* 148b⟩≡ (142)
```
if(ipforme(f, laddr) != Runi) {
    switch(version){
    case V4:
        v4tov6(laddr, ifc->lifc->local);
        break;
    ⟨udpiput() new conv to create, ipv6 case 516b⟩
    default:
        panic("udpiput3: version %d", version);
    }
}
```
Uses Runi 38e, V4 21g, ipforme() 125e, and v4tov6() 21c.

11.8.3 User level UDP headers

⟨*struct* Udpcb 148c⟩≡ (405)
```
struct Udpcb
{
    QLock;
    ⟨Idpcb other fields 148d⟩
};
```

⟨Idpcb *other fields* 148d⟩≡ (148c)
```
uchar   headers;
```

⟨*function* udpctl 148e⟩≡ (405)
```
char*
udpctl(Conv *c, char **f, int n)
{
    Udpcb *ucb;

    ucb = (Udpcb*)c->ptcl;
    if(n == 1){
        if(strcmp(f[0], "headers") == 0){
            ucb->headers = 7;   /* new headers format */
            return nil;
        }
    }
    return "unknown control request";
}
```

⟨*constant* UDP_USEAD7 148f⟩≡ (403b)
```
UDP_USEAD7  = 52,
```

⟨udpkick() *locals* 148g⟩≡ (140b) 513c▷
```
Udpcb *ucb;
ipaddr laddr, raddr;
ushort rport;
```

⟨udpkick() *special headers processing* 149a⟩≡ (140b)
```
  ucb = (Udpcb*)c->ptcl;
  switch(ucb->headers) {
  case 7:
      /* get user specified addresses */
      bp = pullupblock(bp, UDP_USEAD7);
      if(bp == nil)
          return;
      ipmove(raddr, bp->rp);
      bp->rp += IPaddrlen;
      ipmove(laddr, bp->rp);
      bp->rp += IPaddrlen;

      /* pick interface closest to dest */
      if(ipforme(f, laddr) != Runi)
          findlocalip(f, laddr, raddr);
      bp->rp += IPaddrlen;        /* Ignore ifc address */
      rport = nhgets(bp->rp);
      bp->rp += 2+2;          /* Ignore local port */
      break;
  default:
      rport = 0;
      break;
  }
```
Uses `IPaddrlen` 20c, `Runi` 38e, `UDP_USEAD7-372` 148f, `findlocalip()` 379, `ipforme()` 125e, and `ipmove` 23c.

⟨udpkick() *if special headers* 149b⟩≡ (140b)
```
  if(ucb->headers) {
      v6tov4(uh4->udpdst, raddr);
      hnputs(uh4->udpdport, rport);
      v6tov4(uh4->udpsrc, laddr);
      rc = nil;
  }
```
Uses `v6tov4()` 21d.

⟨udpiput() *if special headers* 149c⟩≡ (142)
```
  switch(ucb->headers){
  case 7:
      /* pass the src address */
      bp = padblock(bp, UDP_USEAD7);
      p = bp->rp;
      ipmove(p, raddr); p += IPaddrlen;
      ipmove(p, laddr); p += IPaddrlen;
      ipmove(p, ifc->lifc->local); p += IPaddrlen;
      hnputs(p, rport); p += 2;
      hnputs(p, lport);
      break;
  }
```
Uses `IPaddrlen` 20c, `UDP_USEAD7-372` 148f, and `ipmove` 23c.

Chapter 12

Reliable Communication: IL

IL (Internet Link) is Plan 9's lightweight reliable transport protocol, simpler than TCP. It provides reliable, sequenced delivery with congestion control, but without TCP's complexity (no sliding windows, no Nagle algorithm, no slow start). IL was designed for the local area networks common in Bell Labs, where packet loss is rare and latency is low. It has since been deprecated in favor of TCP for wider compatibility, but the code remains as an elegant example of a minimal reliable protocol.

12.1 Protocol initialisation

`ilinit` registers IL as protocol 40. Like UDP, it fills in a `Proto` method table, but IL also needs `inuse` (because connections have state that prevents reuse) and timing infrastructure (`inittimescale`) for its adaptive retransmission. IL creates per-conversation control blocks (`Ilcb`) that track the state machine, sequence numbers, and round-trip time estimates.

⟨*function* `ilinit` 150⟩≡ (401b)

```
void
ilinit(Fs *f)
{
    Proto *il;

    inittimescale();

    il = smalloc(sizeof(Proto));
    il->priv = smalloc(sizeof(Ilpriv));

    il->name = "il";
    il->create = ilcreate;
    il->close = ilclose;

    il->connect = ilconnect;
    il->announce = ilannounce;
    il->ctl = nil;

    il->rcv = iliput;

    il->state = ilstate;
    il->stats = ilxstats;

    il->inuse = ilinuse;
    il->advise = iladvise;

    il->gc = nil;
    il->ipproto = IP_ILPROTO;
```

```
    il->nc = scalednconv();
    il->ptclsize = sizeof(Ilcb);

    Fsproto(f, il);
}
```

Uses Fsproto() 68a, IP_ILPROTO-77 151a, iladvise() 400b, ilannounce() 156d, ilclose() 166b, ilconnect() 156c, ilcreate() 155c, ilinuse() 152c, iliput() 158, ilstate() 168b, ilxstats() 168a, inittimescale() 151b, and scalednconv() 151d.

⟨*constant* IP_ILPROTO 151a⟩≡ (392b)

```
IP_ILPROTO = 40,
```

⟨*function* inittimescale 151b⟩≡ (401b)

```
/* calculate scale constants that converts fast ticks to ms (more or less) */
static void
inittimescale(void)
{
    uvlong hz;

    arch_fastticks(&hz);
    if(hz > 1000){
        scalediv = hz/1000;
        scalemul = 1;
    } else {
        scalediv = 1;
        scalemul = 1000/hz;
    }
}
```

Uses scalediv-92 151c and scalemul-93 151c.

⟨*global* scalexxx 151c⟩≡ (401b)

```
static ulong scalediv, scalemul;
```

⟨*function* scalednconv 151d⟩≡ (331c)

```
ulong
scalednconv(void)
{
    if(cpuserver && conf.npage*BY2PG >= 128*MB)
        return Nchans*4;
    return Nchans;
}
```

Uses Nchans 135b.

12.2 Protocol data structures

IL's state machine has 6 states: Ilclosed, Ilsyncer (initiating a connection), Ilsyncee (responding to a connection), Ilestablished (data transfer), Illistening (waiting for connections), and Ilclosing. The protocol uses 7 packet types including Ilsync (connection request), Ildata, Ilack, and Ilclose. The Ilcb (control block) tracks sequence numbers, acknowledgment state, and adaptive retransmission timers.

12.2.1 IL state

⟨*enum* _anon_ (kernel/network/ip/il.c) 152a⟩≡ (401b)

```
enum il_state    /* Connection state */
{
    Ilclosed,

    Ilsyncer,
    Ilsyncee,
    Ilestablished,
    Illistening,
    Ilclosing,

    Ilopening,  /* only for file server */
};
```

⟨*global* ilstates 152b⟩≡ (401b)

```
char *ilstates[] =
{
    "Closed",
    "Syncer",
    "Syncee",
    "Established",
    "Listen",
    "Closing",
    "Opening",  /* only for file server */
};
```

⟨*function* ilinuse 152c⟩≡ (401b)

```
static int
ilinuse(Conv *c)
{
    Ilcb *ic;

    ic = (Ilcb*)(c->ptcl);
    return ic->state != Ilclosed;

}
```

Uses Ilclosed-45 152a.

12.2.2 IL control packet type

⟨*enum* _anon_ (kernel/network/ip/il.c)2 152d⟩≡ (401b)

```
enum    /* Packet types */
{
    Ilsync,
    Ildata,
    Ildataquery,
    Ilack,
    Ilquery,
    Ilstate,
    Ilclose,
};
```

⟨*global* iltype 152e⟩≡ (401b)

```
char *iltype[] =
{
    "sync",
    "data",
```

```
    "dataquery",
    "ack",
    "query",
    "state",
    "close"
};
```

12.2.3 `Ilcb`

The `Ilcb` (IL control block) is the per-conversation state for IL. It tracks the state machine, sequence numbers (`next` for sending, `recvd` for receiving), unacknowledged packets (a linked list of `Block`s awaiting ACKs), and out-of-order packets. The adaptive retransmission section tracks round-trip time statistics: `delay` (average RTT), `mdev` (mean deviation), and `maxrtt` (peak), which are used to set the retransmission timeout dynamically—packets to a slow host get longer timeouts while fast local hosts get tight ones.

⟨*struct* `Ilcb` 153⟩≡ (401b)

```
struct Ilcb   /* Control block */
{
    // enum<il_state>
    int state;  /* Connection state */

    Conv *conv;

    QLock ackq;  /* Unacknowledged queue */
    Block *unacked;
    Block *unackedtail;
    ulong unackedbytes;
    QLock outo;  /* Out of order packet queue */
    Block *outoforder;
    ulong next;  /* Id of next to send */
    ulong recvd;  /* Last packet received */
    ulong acksent; /* Last packet acked */
    ulong start;  /* Local start id */
    ulong rstart;  /* Remote start id */
    int window;  /* Maximum receive window */
    int rxquery; /* number of queries on this connection */
    int rxtot;  /* number of retransmits on this connection */
    int rexmit;  /* number of retransmits of *unacked */
    ulong qt[Nqt+1]; /* state table for query messages */
    int qtx;  /* ... index into qt */

    /* if set, fasttimeout causes a connection request to terminate after 4*Iltickms */
    int fasttimeout;

    /* timers */
    ulong lastxmit; /* time of last xmit */
    ulong lastrecv; /* time of last recv */
    ulong timeout; /* retransmission time for *unacked */
    ulong acktime; /* time to send next ack */
    ulong querytime; /* time to send next query */

    /* adaptive measurements */
    int delay;  /* Average of the fixed rtt delay */
    int rate;  /* Average uchar rate */
    int mdev;  /* Mean deviation of rtt */
    int maxrtt;  /* largest rtt seen */
    ulong rttack;  /* The ack we are waiting for */
    int rttlen;  /* Length of rttack packet */
```

```
    uvlong rttstart; /* Time we issued rttack packet */
};
```
Uses Nqt-74 392a.

12.2.4 Ilpriv

⟨*struct* Ilpriv 154a⟩≡ (401b)
```
struct Ilpriv
{
    Ipht ht;

    ulong dup;
    ulong dupb;

    ⟨Ilpriv stat fields 154b⟩

    /* keeping track of the ack kproc */
    int ackprocstarted;
    QLock apl;
};
```

12.2.5 Statistics

⟨Ilpriv *stat fields* 154b⟩≡ (154a) 154c ▷
```
ulong csumerr;  /* checksum errors */
ulong hlenerr;  /* header length error */
ulong lenerr;   /* short packet */
ulong order;   /* out of order */
ulong rexmit;   /* retransmissions */
```

⟨Ilpriv *stat fields* 154c⟩+≡ (154a) ◁ 154b
```
// map<enum<il_stat>, uvlong>
uvlong stats[Nstats];
```
Uses Nstats-90 154d.

⟨*enum* _anon_ (kernel/network/ip/il.c)6 154d⟩≡ (401b)
```
enum il_stat
{
    InMsgs,
    OutMsgs,
    CsumErrs,  /* checksum errors */
    HlenErrs,  /* header length error */
    LenErrs,  /* short packet */
    OutOfOrder,  /* out of order */
    Retrans,  /* retransmissions */
    DupMsg,
    DupBytes,
    DroppedMsgs,

    Nstats,
};
```

⟨*global* statnames*(*(kernel/network/ip/il.c)*)* 154e⟩≡ (401b)
```
static char *statnames[] =
{
[InMsgs] "InMsgs",
[OutMsgs] "OutMsgs",
```

```
[CsumErrs] "CsumErrs",
[HlenErrs] "HlenErr",
[LenErrs] "LenErrs",
[OutOfOrder] "OutOfOrder",
[Retrans] "Retrans",
[DupMsg] "DupMsg",
[DupBytes] "DupBytes",
[DroppedMsgs] "DroppedMsgs",
};
```

12.3 Protocol header

The IL header sits on top of the IP header and adds 18 bytes: source and destination ports (like UDP), a packet length, a packet type (sync, data, ack, etc.), a 32-bit sequence number (`ilid`), and a 32-bit acknowledgment number (`ilack`). The sequence and ack numbers are the core of IL's reliable delivery—each side tracks what it has sent and what the other side has received.

⟨*struct* `Ilhdr` 155a⟩≡ (401b)

```
struct Ilhdr
{
    /* ip header */
    uchar vihl;  /* Version and header length */
    uchar tos;  /* Type of service */
    uchar length[2]; /* packet length */
    uchar id[2];  /* Identification */
    uchar frag[2]; /* Fragment information */
    uchar ttl;  /* Time to live */

    uchar proto;  /* Protocol */
    uchar cksum[2]; /* Header checksum */
    ipv4 src;  /* Ip source */
    ipv4 dst;  /* Ip destination */

    /* IL header */
    uchar ilsum[2]; /* Checksum including header */
    uchar illen[2]; /* Packet length */
    uchar iltype;  /* Packet type */
    uchar ilspec;  /* Special */
    uchar ilsrc[2]; /* Src port */
    uchar ildst[2]; /* Dst port */
    uchar ilid[4]; /* Sequence id */
    uchar ilack[4]; /* Acked sequence */
};
```

⟨*constant* `IL_xxxSIZE` 155b⟩≡ (392b)

```
IL_IPSIZE  = 20,
IL_HDRSIZE = 18,
```

12.4 /net/il/clone

`ilcreate` sets up an IL conversation with message-oriented queues (like UDP) plus an acknowledgment processing goroutine. Unlike UDP, IL needs a "kick" function (`ilkick`) that adds sequence numbers and saves sent packets for retransmission.

⟨*function* `ilcreate` 155c⟩≡ (401b)

```
static void
```

```
ilcreate(Conv *c)
{
    c->rq = qopen(Maxrq, 0, 0, c);
    c->wq = qbypass(ilkick, c);
}
```
Uses `Maxrq-73` 156a and `ilkick()` 157.

⟨*constant Maxrq(IL)* 156a⟩≡ (391b)
```
Maxrq  = 64*1024,
```

12.5 /net/il/x/ctl

IL's "connect" is fundamentally different from UDP's: it calls `ilstart` with `IL_CONNECT`, which sends an `Ilsync` packet and transitions to `Ilsyncer` state. The connection is not established until the remote side replies with its own `Ilsync`. "Announce" calls `ilstart` with `IL_LISTEN`, which transitions to `Illistening` and waits for incoming sync packets.

⟨*enum* `mode`*(*`(kernel/network/ip/il.c)`*)* 156b⟩≡ (401b)
```
enum mode {
    IL_LISTEN = 0,
    IL_CONNECT = 1,
};
```

12.5.1 Connect

⟨*function* `ilconnect` 156c⟩≡ (401b)
```
static char*
ilconnect(Conv *c, char **argv, int argc)
{
    char *err;
    char *p;
    bool fast = false;

    ⟨ilconnect() set fast 168c⟩

    err = Fsstdconnect(c, argv, argc);
    if(err != nil)
        return err;
    return ilstart(c, IL_CONNECT, fast);
}
```
Uses `Fsstdconnect()` 56b, `IL_CONNECT-79` 156b, and `ilstart()` 165.

12.5.2 Announce

⟨*function* `ilannounce` 156d⟩≡ (401b)
```
/* called with c locked */
static char*
ilannounce(Conv *c, char **argv, int argc)
{
    char *err;

    err = Fsstdannounce(c, argv, argc);
    if(err != nil)
        return err;
    err = ilstart(c, IL_LISTEN, false);
```

```
    if(err != nil)
        return err;
    Fsconnected(c, nil);

    return nil;
}
```

Uses `Fsstdannounce()` 58b, `IL_LISTEN-78` 156b, and `ilstart()` 165.

12.6 IO

`ilkick` is the write-side callback: it prepends the IL header (including sequence number, ack number, and packet type), computes the checksum, and calls `ipoput4` to send the packet. Sent packets are saved in `ic->unacked` for possible retransmission. `iliput` is the read-side handler: it validates incoming packets, dispatches based on state and packet type to `ilprocess`, delivers data packets to the conversation's read queue, and sends acknowledgments.

12.6.1 Writing: `ilkick()`

⟨*function* `ilkick` 157⟩≡ (401b)

```
void
ilkick(void *x, Block *bp)
{
    Conv *c = x;
    Ilhdr *ih;
    Ilcb *ic;
    int dlen;
    ulong id, ack;
    Fs *f;
    Ilpriv *priv;

    f = c->p->f;
    priv = c->p->priv;
    ic = (Ilcb*)c->ptcl;

    if(bp == nil)
        return;

    switch(ic->state) {
    case Ilclosed:
    case Illistening:
    case Ilclosing:
        freeblist(bp);
        qhangup(c->rq, nil);
        return;
    }

    dlen = blocklen(bp);

    /* Make space to fit il & ip */
    bp = padblock(bp, IL_IPSIZE+IL_HDRSIZE);
    ih = (Ilhdr *)(bp->rp);
    ih->vihl = IP_VER4;

    /* Ip fields */
    ih->frag[0] = 0;
    ih->frag[1] = 0;
```

```
    v6tov4(ih->dst, c->raddr);
    v6tov4(ih->src, c->laddr);
    ih->proto = IP_ILPROTO;

    /* Il fields */
    hnputs(ih->illen, dlen+IL_HDRSIZE);
    hnputs(ih->ilsrc, c->lport);
    hnputs(ih->ildst, c->rport);

    qlock(&ic->ackq);
    id = ic->next++;
    hnputl(ih->ilid, id);
    ack = ic->recvd;
    hnputl(ih->ilack, ack);
    ic->acksent = ack;
    ic->acktime = NOW + AckDelay;
    ih->iltype = Ildata;
    ih->ilspec = 0;
    ih->ilsum[0] = 0;
    ih->ilsum[1] = 0;

    /* Checksum of ilheader plus data (not ip & no pseudo header) */
    if(ilcksum)
        hnputs(ih->ilsum, ptclcsum(bp, IL_IPSIZE, dlen+IL_HDRSIZE));

    ilackq(ic, bp);

    qunlock(&ic->ackq);

    /* Start the round trip timer for this packet if the timer is free */
    if(ic->rttack == 0) {
        ic->rttack = id;
        ic->rttstart = arch_fastticks(nil);
        ic->rttlen = dlen + IL_IPSIZE + IL_HDRSIZE;
    }

    if(later(NOW, ic->timeout, nil))
        ilsettimeout(ic);

    // Let's go, let's send the data
    ipoput4(f, bp, false, c->ttl, c->tos, c);

    priv->stats[OutMsgs]++;
}
```

Uses AckDelay-61 391b, IL_HDRSIZE-76 155b, IL_IPSIZE-75 155b, IP_ILPROTO-77 151a, IP_VER4 94a, Ilclosed-45 152a, Ilclosing-50 152a, Ildata-53 152d, Illistening-49 152a, NOW 234a, OutMsgs-81 154d, ilackq() 392d, ilcksum 167b, ilsettimeout() 397b, ipoput4() 93, later() 398c, ptclcsum() 90d, and v6tov4() 21d.

12.6.2 Reading: iliput()

⟨*function* iliput 158⟩≡ (401b)

```
void
iliput(Proto *il, Ipifc*, Block *bp)
{
    char *st;
    Ilcb *ic;
    Ilhdr *ih;
    ipaddr raddr, laddr;
    ushort sp, dp, csum;
```

```
	int plen, illen;
	Conv *new, *s;
	Ilpriv *ipriv;

	ipriv = il->priv;

	ih = (Ilhdr *)bp->rp;
	plen = blocklen(bp);
	if(plen < IL_IPSIZE+IL_HDRSIZE){
		//netlog(il->f, Logil, "il: hlenerr\n");
		ipriv->stats[HlenErrs]++;
		goto raise;
	}

	illen = nhgets(ih->illen);
	if(illen+IL_IPSIZE > plen){
		//netlog(il->f, Logil, "il: lenerr\n");
		ipriv->stats[LenErrs]++;
		goto raise;
	}

	sp = nhgets(ih->ildst);
	dp = nhgets(ih->ilsrc);
	v4tov6(raddr, ih->src);
	v4tov6(laddr, ih->dst);

	if((csum = ptclcsum(bp, IL_IPSIZE, illen)) != 0) {
		if(ih->iltype > Ilclose)
			st = "?";
		else
			st = iltype[ih->iltype];
		ipriv->stats[CsumErrs]++;
		//netlog(il->f, Logil, "il: cksum %ux %s, pkt(%ux id %ud ack %I/%d->%d)\n",
//	csum, st, nhgetl(ih->ilid), nhgetl(ih->ilack), raddr, sp, dp);
		USED(csum); USED(st);
		goto raise;
	}

	qlock(il);
	s = iphtlook(&ipriv->ht, raddr, dp, laddr, sp);
	if(s == nil){
		if(ih->iltype == Ilsync)
			ilreject(il->f, ih);	/* no listener */
		qunlock(il);
		goto raise;
	}

	ic = (Ilcb*)s->ptcl;
	if(ic->state == Illistening){
		if(ih->iltype != Ilsync){
			qunlock(il);
			if(ih->iltype > Ilclose)
				st = "?";
			else
				st = iltype[ih->iltype];
			ilreject(il->f, ih);	/* no channel and not sync */
			//netlog(il->f, Logil, "il: no channel, pkt(%s id %ud ack %ud %I/%ud->%ud)\n",
//	st, nhgetl(ih->ilid), nhgetl(ih->ilack), raddr, sp, dp);
			USED(st);
			goto raise;
```

```
        }

        new = Fsnewcall(s, raddr, dp, laddr, sp, V4);
        if(new == nil){
            qunlock(il);
            //netlog(il->f, Logil, "il: bad newcall %I/%ud->%ud\n", raddr, sp, dp);
            ilsendctl(s, ih, Ilclose, 0, nhgetl(ih->ilid), 0);
            goto raise;
        }
        s = new;

        ic = (Ilcb*)s->ptcl;

        ic->conv = s;
        ic->state = Ilsyncee;
        ilcbinit(ic);
        ic->rstart = nhgetl(ih->ilid);
        iphtadd(&ipriv->ht, s);
    }

    qlock(s);
    qunlock(il);
    if(waserror()){
        qunlock(s);
        nexterror();
    }

    ilprocess(s, ih, bp);

    qunlock(s);
    poperror();
    return;
raise:
    freeblist(bp);
}
```

Uses CsumErrs-82 154d, Fsnewcall() 144, HlenErrs-83 154d, IL_HDRSIZE-76 155b, IL_IPSIZE-75 155b, Ilclose-58 152d, Illistening-49 152a, Ilsync-52 152d, Ilsyncee-47 152a, LenErrs-84 154d, V4 21g, ilcbinit() 166a, ilprocess() 162a, ilreject() 397a, ilsendctl() 160, iltype 152e, iphtadd() 136d, iphtlook() 143, ptclcsum() 90d, and v4tov6() 21c.

12.7 State machine

IL uses a simple two-way handshake: the syncer sends an `Ilsync` packet and the syncee replies with its own `Ilsync`. Once both sides have exchanged sync packets with sequence numbers, the connection moves to `Ilestablished`. `ilprocess` is the main state machine dispatcher—a large switch on `ic->state` and packet type that handles acks, data delivery, retransmission requests (`Ilquery`/`Ildataquery`), and close sequencing. `ilsendctl` sends control packets (sync, ack, close) without user data.

12.7.1 ilsendctl()

⟨*function* ilsendctl 160⟩≡ (401b)

```
void
ilsendctl(Conv *ipc, Ilhdr *inih, int type, ulong id, ulong ack, int ilspec)
{
    Ilhdr *ih;
    Ilcb *ic;
    Block *bp;
```

```
    int ttl, tos;

    bp = allocb(IL_IPSIZE+IL_HDRSIZE);
    bp->wp += IL_IPSIZE+IL_HDRSIZE;

    ih = (Ilhdr *)(bp->rp);
    ih->vihl = IP_VER4;

    /* Ip fields */
    ih->proto = IP_ILPROTO;
    hnputs(ih->illen, IL_HDRSIZE);
    ih->frag[0] = 0;
    ih->frag[1] = 0;
    if(inih) {
        hnputl(ih->dst, nhgetl(inih->src));
        hnputl(ih->src, nhgetl(inih->dst));
        hnputs(ih->ilsrc, nhgets(inih->ildst));
        hnputs(ih->ildst, nhgets(inih->ilsrc));
        hnputl(ih->ilid, nhgetl(inih->ilack));
        hnputl(ih->ilack, nhgetl(inih->ilid));
        ttl = MAXTTL;
        tos = DFLTTOS;
    }
    else {
        v6tov4(ih->dst, ipc->raddr);
        v6tov4(ih->src, ipc->laddr);
        hnputs(ih->ilsrc, ipc->lport);
        hnputs(ih->ildst, ipc->rport);
        hnputl(ih->ilid, id);
        hnputl(ih->ilack, ack);
        ic = (Ilcb*)ipc->ptcl;
        ic->acksent = ack;
        ic->acktime = NOW;
        ttl = ipc->ttl;
        tos = ipc->tos;
    }
    ih->iltype = type;
    ih->ilspec = ilspec;
    ih->ilsum[0] = 0;
    ih->ilsum[1] = 0;

    if(ilcksum)
        hnputs(ih->ilsum, ptclcsum(bp, IL_IPSIZE, IL_HDRSIZE));

if(ipc==nil)
    panic("ipc is nil caller is %#p", getcallerpc(&ipc));
if(ipc->p==nil)
    panic("ipc->p is nil");

    //netlog(ipc->p->f, Logilmsg, "ctl(%s id %d ack %d %d->%d)\n",
    //  iltype[ih->iltype], nhgetl(ih->ilid), nhgetl(ih->ilack),
    //  nhgets(ih->ilsrc), nhgets(ih->ildst));

    ipoput4(ipc->p->f, bp, false, ttl, tos, ipc);
}
```

Uses DFLTTOS 232b, IL_HDRSIZE-76 155b, IL_IPSIZE-75 155b, IP_ILPROTO-77 151a, IP_VER4 94a, MAXTTL 81g, NOW 234a, ilcksum 167b, ipoput4() 93, ptclcsum() 90d, and v6tov4() 21d.

12.7.2 ilprocess()

⟨*function* ilprocess 162a⟩≡ (401b)
```
/* DEBUG */
void
ilprocess(Conv *s, Ilhdr *h, Block *bp)
{
    Ilcb *ic;

    ic = (Ilcb*)s->ptcl;

    USED(ic);
    //netlog(s->p->f, Logilmsg, "%11s rcv %lud/%lud snt %lud/%lud pkt(%s id %d ack %ud %ud->%ud) ",
//  ilstates[ic->state],  ic->rstart, ic->recvd, ic->start,
//  ic->next, iltype[h->iltype], nhgetl(h->ilid),
//  nhgetl(h->ilack), nhgets(h->ilsrc), nhgets(h->ildst));

    _ilprocess(s, h, bp);

    //netlog(s->p->f, Logilmsg, "%11s rcv %lud snt %lud\n", ilstates[ic->state], ic->recvd, ic->next);
}
```
Uses _ilprocess() 162b.

⟨*function* _ilprocess 162b⟩≡ (401b)
```
void
_ilprocess(Conv *s, Ilhdr *h, Block *bp)
{
    Ilcb *ic;
    ulong id, ack;
    Ilpriv *priv;

    id = nhgetl(h->ilid);
    ack = nhgetl(h->ilack);

    ic = (Ilcb*)s->ptcl;

    ic->lastrecv = NOW;
    ic->querytime = NOW + QueryTime;
    priv = s->p->priv;
    priv->stats[InMsgs]++;

    switch(ic->state) {
    default:
        //netlog(s->p->f, Logil, "il: unknown state %d\n", ic->state);
    case Ilclosed:
        freeblist(bp);
        break;
    case Ilsyncer:
        switch(h->iltype) {
        default:
            break;
        case Ilsync:
            if(ack != ic->start)
                ilhangup(s, "connection rejected");
            else {
                ic->recvd = id;
                ic->rstart = id;
                ilsendctl(s, nil, Ilack, ic->next, ic->recvd, 0);
                ic->state = Ilestablished;
                ic->fasttimeout = 0;
```

```
			ic->rexmit = 0;
			Fsconnected(s, nil);
			ilpullup(s);
		}
		break;
	case Ilclose:
		if(ack == ic->start)
			ilhangup(s, "connection rejected");
		break;
	}
	freeblist(bp);
	break;
case Ilsyncee:
	switch(h->iltype) {
	default:
		break;
	case Ilsync:
		if(id != ic->rstart || ack != 0){
			illocalclose(s);
		} else {
			ic->recvd = id;
			ilsendctl(s, nil, Ilsync, ic->start, ic->recvd, 0);
		}
		break;
	case Ilack:
		if(ack == ic->start) {
			ic->state = Ilestablished;
			ic->fasttimeout = 0;
			ic->rexmit = 0;
			ilpullup(s);
		}
		break;
	case Ildata:
		if(ack == ic->start) {
			ic->state = Ilestablished;
			ic->fasttimeout = 0;
			ic->rexmit = 0;
			goto established;
		}
		break;
	case Ilclose:
		if(ack == ic->start)
			ilhangup(s, "remote close");
		break;
	}
	freeblist(bp);
	break;
case Ilestablished:
established:
	switch(h->iltype) {
	case Ilsync:
		if(id != ic->rstart)
			ilhangup(s, "remote close");
		else
			ilsendctl(s, nil, Ilack, ic->next, ic->rstart, 0);
		freeblist(bp);
		break;
	case Ildata:
		/*
		 * avoid consuming all the mount rpc buffers in the
```

```
         * system.  if the input queue is too long, drop this
         * packet.
         */
        if (s->rq && qlen(s->rq) >= Maxrq) {
            priv->stats[DroppedMsgs]++;
            freeblist(bp);
            break;
        }

        ilackto(ic, ack, bp);
        iloutoforder(s, h, bp);
        ilpullup(s);
        break;
    case Ildataquery:
        ilackto(ic, ack, bp);
        iloutoforder(s, h, bp);
        ilpullup(s);
        ilsendctl(s, nil, Ilstate, ic->next, ic->recvd, h->ilspec);
        break;
    case Ilack:
        ilackto(ic, ack, bp);
        freeblist(bp);
        break;
    case Ilquery:
        ilackto(ic, ack, bp);
        ilsendctl(s, nil, Ilstate, ic->next, ic->recvd, h->ilspec);
        freeblist(bp);
        break;
    case Ilstate:
        if(ack >= ic->rttack)
            ic->rttack = 0;
        ilackto(ic, ack, bp);
        if(h->ilspec > Nqt)
            h->ilspec = 0;
        if(ic->qt[h->ilspec] > ack){
            ilrexmit(ic);
            ilsettimeout(ic);
        }
        freeblist(bp);
        break;
    case Ilclose:
        freeblist(bp);
        if(ack < ic->start || ack > ic->next)
            break;
        ic->recvd = id;
        ilsendctl(s, nil, Ilclose, ic->next, ic->recvd, 0);
        ic->state = Ilclosing;
        ilsettimeout(ic);
        ilfreeq(ic);
        break;
    }
    break;
case Illistening:
    freeblist(bp);
    break;
case Ilclosing:
    switch(h->iltype) {
    case Ilclose:
        ic->recvd = id;
        ilsendctl(s, nil, Ilclose, ic->next, ic->recvd, 0);
```

```
            if(ack == ic->next)
                ilhangup(s, nil);
            break;
        default:
            break;
        }
        freeblist(bp);
        break;
    }
}
```

Uses DroppedMsgs-89 154d, Ilack-55 152d, Ilclose-58 152d, Ilclosed-45 152a, Ilclosing-50 152a, Ildata-53 152d, Ildataquery-54 152d, Ilestablished-48 152a, Illistening-49 152a, Ilquery-56 152d, Ilstate-57 152d, Ilsync-52 152d, Ilsyncee-47 152a, Ilsyncer-46 152a, InMsgs-80 154d, Maxrq-73 156a, NOW 234a, Nqt-74 392a, QueryTime-63 391b, ilackto() 393, ilfreeq() 400a, ilhangup() 395a, illocalclose() 167a, iloutoforder() 396, ilpullup() 395b, ilrexmit() 394, ilsendctl() 160, and ilsettimeout() 397b.

12.7.3 Start

⟨*function* ilstart 165⟩≡ (401b)

```
char*
ilstart(Conv *c, int type, bool fasttimeout)
{
    Ilcb *ic;
    Ilpriv *ipriv;
    char kpname[KNAMELEN];

    ipriv = c->p->priv;

    if(ipriv->ackprocstarted == 0){
        qlock(&ipriv->apl);
        if(ipriv->ackprocstarted == 0){
            sprint(kpname, "#I%dilack", c->p->f->dev);
            kproc(kpname, ilackproc, c->p);
            ipriv->ackprocstarted = 1;
        }
        qunlock(&ipriv->apl);
    }

    ic = (Ilcb*)c->ptcl;
    ic->conv = c;

    if(ic->state != Ilclosed)
        return nil;

    ilcbinit(ic);

    if(fasttimeout){
        /* timeout if we can't connect quickly */
        ic->fasttimeout = 1;
        ic->timeout = NOW+Iltickms;
        ic->rexmit = MaxRexmit - 4;
    };

    switch(type) {
    case IL_CONNECT:
        ic->state = Ilsyncer;
        iphtadd(&ipriv->ht, c);
        ilsendctl(c, nil, Ilsync, ic->start, ic->recvd, 0);
        break;
```

```
    case IL_LISTEN:
        ic->state = Illistening;
        iphtadd(&ipriv->ht, c);
        break;
    default:
        //netlog(c->p->f, Logil, "il: start: type %d\n", type);
        break;
    }

    return nil;
}
```

Uses IL_CONNECT-79 156b, IL_LISTEN-78 156b, Ilclosed-45 152a, Illistening-49 152a, Ilsync-52 152d, Ilsyncer-46 152a, Iltickms-60 391b, MaxRexmit-65 391b, NOW 234a, ilcbinit() 166a, ilsendctl() 160, and iphtadd() 136d.

⟨*function* ilcbinit 166a⟩≡ (401b)

```
void
ilcbinit(Ilcb *ic)
{
    ic->start = nrand(0x1000000);
    ic->next = ic->start+1;
    ic->recvd = 0;
    ic->window = Defaultwin;
    ic->unackedbytes = 0;
    ic->unacked = nil;
    ic->outoforder = nil;
    ic->rexmit = 0;
    ic->rxtot = 0;
    ic->rxquery = 0;
    ic->qtx = 1;
    ic->fasttimeout = 0;

    /* timers */
    ic->delay = DefRtt<<LogAGain;
    ic->mdev = DefRtt<<LogDGain;
    ic->rate = DefByteRate<<LogAGain;
    ic->querytime = NOW + QueryTime;
    ic->lastrecv = NOW; /* or we'll timeout right away */
    ilsettimeout(ic);
}
```

Uses DefByteRate-71 391b, DefRtt-72 391b, Defaultwin-66 391b, LogAGain-67 391b, LogDGain-69 391b, NOW 234a, QueryTime-63 391b, and ilsettimeout() 397b.

12.7.4 Close

⟨*function* ilclose 166b⟩≡ (401b)

```
static void
ilclose(Conv *c)
{
    Ilcb *ic;

    ic = (Ilcb*)c->ptcl;

    qclose(c->rq);
    qclose(c->wq);
    qclose(c->eq);

    switch(ic->state) {
    case Ilclosing:
```

```
        case Ilclosed:
            break;
        case Ilsyncer:
        case Ilsyncee:
        case Ilestablished:
            ic->state = Ilclosing;
            ilsettimeout(ic);
            ilsendctl(c, nil, Ilclose, ic->next, ic->recvd, 0);
            break;
        case Illistening:
            illocalclose(c);
            break;
        }
        ilfreeq(ic);
    }
```

Uses Ilclose-58 152d, Ilclosed-45 152a, Ilclosing-50 152a, Ilestablished-48 152a, Illistening-49 152a, Ilsyncee-47 152a, Ilsyncer-46 152a, ilfreeq() 400a, illocalclose() 167a, ilsendctl() 160, and ilsettimeout() 397b.

⟨*function* illocalclose 167a⟩≡ (401b)

```
    void
    illocalclose(Conv *c)
    {
        Ilcb *ic;
        Ilpriv *ipriv;

        ipriv = c->p->priv;
        ic = (Ilcb*)c->ptcl;
        ic->state = Ilclosed;
        iphtrem(&ipriv->ht, c);
        ipmove(c->laddr, IPnoaddr);
        c->lport = 0;
    }
```

Uses IPnoaddr 23b, Ilclosed-45 152a, iphtrem() 137a, and ipmove 23c.

12.7.5 Syncer

12.7.6 Syncee

12.7.7 Establised

12.8 Features

12.8.1 Reliable datagram service

⟨*global* ilcksum 167b⟩≡ (401b)

```
    bool  ilcksum = true;
```

Uses ilcksum 167b.

12.8.2 In sequence delivery

12.8.3 Retransmission of lost messages

12.8.4 Out-of-order messages saving

12.8.5 Adaptive timeouts

12.8.6 Keep-alive

12.9 Other files

12.9.1 /net/il/stats

⟨*function* ilxstats 168a⟩≡ (401b)

```
int
ilxstats(Proto *il, char *buf, int len)
{
    Ilpriv *priv;
    char *p, *e;
    int i;

    priv = il->priv;
    p = buf;
    e = p+len;
    for(i = 0; i < Nstats; i++)
        p = seprint(p, e, "%s: %llud\n", statnames[i], priv->stats[i]);
    return p - buf;
}
```

Uses Nstats-90 154d and statnames-91 154e.

12.9.2 /net/il/x/status

⟨*function* ilstate 168b⟩≡ (401b)

```
static int
ilstate(Conv *c, char *state, int n)
{
    Ilcb *ic;

    ic = (Ilcb*)(c->ptcl);
    return snprint(state, n, "%s qin %d qout %d del %5.5d Br %5.5d md %5.5d una %5.5lud rex %5.5d rxq %5.5d max
        ilstates[ic->state],
        c->rq ? qlen(c->rq) : 0,
        c->wq ? qlen(c->wq) : 0,
        ic->delay>>LogAGain, ic->rate>>LogAGain, ic->mdev>>LogDGain,
        ic->unackedbytes, ic->rxtot, ic->rxquery, ic->maxrtt);
}
```

Uses LogAGain-67 391b, LogDGain-69 391b, and ilstates 152b.

12.10 Advanced features

12.10.1 Fast timeout

⟨ilconnect() *set fast* 168c⟩≡ (156c)

```
/* huge hack to quickly try an il connection */
```

```
if(argc > 1){
    p = strstr(argv[1], "!fasttimeout");
    if(p != nil){
        *p = '\0';
        fast = true;
    }
}
```

12.10.2 Special extension

Chapter 13

Standard Reliable Communication: TCP

TCP (Transmission Control Protocol) is the heavy-duty reliable transport: it provides ordered, byte-stream delivery with flow control, congestion avoidance, and retransmission. The implementation is the most complex protocol in the stack, with a state machine (SYN, ESTABLISHED, FIN-WAIT, etc.), sliding windows, round-trip time estimation, and Van Jacobson's congestion control algorithms.

⟨*function* `tcpinit` 170⟩≡ (464)

```
void
tcpinit(Fs *fs)
{
    Proto *tcp;
    Tcppriv *tpriv;

    tcp = smalloc(sizeof(Proto));
    tpriv = tcp->priv = smalloc(sizeof(Tcppriv));

    tcp->name = "tcp";
    tcp->connect = tcpconnect;
    tcp->announce = tcpannounce;
    tcp->ctl = tcpctl;
    tcp->state = tcpstate;
    tcp->create = tcpcreate;
    tcp->close = tcpclose;
    tcp->rcv = tcpiput;
    tcp->advise = tcpadvise;
    tcp->stats = tcpstats;
    tcp->inuse = tcpinuse;
    tcp->gc = tcpgc;

    tcp->ipproto = IP_TCPPROTO;
    tcp->nc = scalednconv();
    tcp->ptclsize = sizeof(Tcpctl);
    tpriv->stats[MaxConn] = tcp->nc;

    Fsproto(fs, tcp);
}
```

Uses `Fsproto()` 68a, `IP_TCPPROTO-269` 406, `MaxConn-331` 410c, `scalednconv()` 151d, `tcpadvise()` 461, `tcpannounce()` 414b, `tcpclose()` 414c, `tcpconnect()` 413a, `tcpcreate()` 417c, `tcpctl()` 462b, `tcpgc()` 462d, `tcpinuse()` 414a, `tcpiput()` 441, `tcpstate()` 413b, and `tcpstats()` 462c.

Chapter 14

Applications

With the protocol stack in place, this chapter surveys the applications built on top of it: remote shell access (Telnet, `cpu`), file transfer (TFTP), and mail. In Plan 9, the most important network application is arguably `cpu`, which exports your local namespace to a remote machine—the opposite of traditional remote login.

14.1 Remote shell: Telnet

14.2 Remote file access: TFTP

14.3 Mail

Chapter 15

Remote Procedure Call: 9P

9P is Plan 9's file protocol—the wire format for all file operations (open, read, write, stat, etc.). Every file server in Plan 9 speaks 9P, and `mount` connects a 9P-speaking server to the local namespace. The marshalling code for 9P messages (`convS2M`, `convM2S`) is in LIBCORE book [Pad16].

Chapter 16

Network File System: `exportfs`

`exportfs` is the logical culmination of Plan 9's "everything is a file" design: it serves a local namespace over the network via 9P, allowing a remote machine to access your files, devices, and even your window system. Combined with `cpu`, which exports *your* namespace to a *remote* CPU, this inverts the traditional client-server model—your local context follows you to any machine on the network.

Chapter 17

Name Resolution: DNS

DNS translates human-readable names (like `google.com`) into IP addresses. In Plan 9, name resolution goes through the connection server (`cs`), which reads the local network database (`/lib/ndb/local`) and queries DNS servers as needed. The `dial` function from Libcore book [Pad16] calls `cs` transparently, so most programs never interact with DNS directly.

Chapter 18

Security

18.1 Denial of service

Chapter 19

Debugging Support

Chapter 20

Profiling Support

Chapter 21

Advanced Topics

This chapter covers advanced networking features that cut across the protocol layers: flow control and congestion avoidance, IPv6 support, broadcast and multicast addressing. The broadcast code adds multiple self-cache entries—subnet broadcast, network broadcast, and the global broadcast address—so that `ipforme` can recognize packets destined for any of these addresses.

21.1 Flow Control

21.2 IPv6

21.3 Broadcast

Broadcast addresses have all host bits set to 1 (e.g., 10.0.2.255 for a /24 network). When adding an IP to an interface, `ipifcadd` registers four broadcast addresses in the self cache: the subnet-directed broadcast, the subnet-directed network address, the classful network broadcast, and the classful network address. For outgoing broadcasts, the gateway is the destination itself (not a router), and the source interface is looked up from the sender's own address.

⟨`ipoput4()` *adjust gate and ifc if broadcast or multicast case* 178a⟩≡ (94c)

```
if(r->type & (Rbcast|Rmulti)) {
    gate = eh->dst;
    sr = v4lookup(f, eh->src, nil);
    if(sr != nil && (sr->type & Runi))
        ifc = sr->ifc;
}
```

Uses `Rbcast` 38e, `Rmulti` 38e, `Runi` 38e, and `v4lookup()` 112a.

⟨`findipifc()` *if broadcast or multicast route* 178b⟩≡ (113e)

```
/* for now for broadcast and multicast, just use first interface */
if(type & (Rbcast|Rmulti)){
    for(cp = f->ipifc->conv; cp < e; cp++){
        if(*cp == 0)
            continue;
        ifc = (Ipifc*)(*cp)->ptcl;
        if(ifc->lifc != nil)
            return ifc;
    }
}
```

Uses `Rbcast` 38e and `Rmulti` 38e.

⟨`ipifcadd()` *locals* 178c⟩≡ (71c) 517e▷

```
ipaddr bcast;
```

⟨ipifcadd() *add broadcast addresses to self cache* 179a⟩≡ (71c)

```
/* add subnet directed broadcast address to the self cache */
for(i = 0; i < IPaddrlen; i++)
    bcast[i] = (ip[i] & mask[i]) | ~mask[i];
addselfcache(f, ifc, lifc, bcast, Rbcast);

/* add subnet directed network address to the self cache */
for(i = 0; i < IPaddrlen; i++)
    bcast[i] = (ip[i] & mask[i]) & mask[i];
addselfcache(f, ifc, lifc, bcast, Rbcast);

/* add network directed broadcast address to the self cache */
memmove(mask, defmask(ip), IPaddrlen);
for(i = 0; i < IPaddrlen; i++)
    bcast[i] = (ip[i] & mask[i]) | ~mask[i];
addselfcache(f, ifc, lifc, bcast, Rbcast);

/* add network directed network address to the self cache */
memmove(mask, defmask(ip), IPaddrlen);
for(i = 0; i < IPaddrlen; i++)
    bcast[i] = (ip[i] & mask[i]) & mask[i];
addselfcache(f, ifc, lifc, bcast, Rbcast);

addselfcache(f, ifc, lifc, IPv4bcast, Rbcast);
```

Uses `IPaddrlen` 20c, `IPv4bcast` 179b, `Rbcast` 38e, `addselfcache()` 127b, and `defmask()` 22b.

⟨*global* IPv4bcast 179b⟩≡ (227e)

```
/*
 *  well known IP addresses
 */
ipaddr IPv4bcast = {
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0xff, 0xff,

    0xff, 0xff, 0xff, 0xff
};
```

21.4 Multicast

Multicast allows one packet to reach multiple machines simultaneously. Each `Conv` can join multicast groups by writing to its `ctl` file. The stack maintains two parallel data structures: `Ipmulti` (per-conversation list of joined groups) and `Ipmcast` (per-interface list of active multicast addresses). When a conversation joins a group, the medium's `addmulti` method configures the hardware (e.g., the Ethernet card's hash filter) to accept frames for that multicast MAC address.

⟨*struct* Ipmulti 179c⟩≡ (234b)

```
/*
 *  one per multicast-lifc pair used by a Conv
 */
struct Ipmulti
{
  ipaddr ma;
  ipaddr ia;
  Ipmulti *next;
};
```

⟨*struct* Ipmcast 180a⟩≡ (384b)
```
struct Ipmcast
{
    Ipmcast *next;
    ipaddr   ma;  /* multicast address */
    ipaddr   ia;  /* interface address */
};
```

⟨ipwrite() *locals* 180b⟩≡ (51c)
```
ipaddr ia, ma;
```

⟨ipwrite() *Qctl case, else if other string* 180c⟩+≡ (55a) ◁148a 180d▷
```
else if(strcmp(cb->f[0], "addmulti") == 0){
    if(cb->nf < 2)
        error("addmulti needs interface address");
    if(cb->nf == 2){
        if(!ipismulticast(cv->raddr))
            error("addmulti for a non multicast address");
        if (parseip(ia, cb->f[1]) == -1)
            error(Ebadip);
        ipifcaddmulti(cv, cv->raddr, ia);
    } else {
        if (parseip(ia, cb->f[1]) == -1 ||
            parseip(ma, cb->f[2]) == -1)
            error(Ebadip);
        if(!ipismulticast(ma))
            error("addmulti for a non multicast address");
        ipifcaddmulti(cv, ma, ia);
    }
}
```
Uses ipifcaddmulti() 382a, ipismulticast() 381c, and parseip() 89.

⟨ipwrite() *Qctl case, else if other string* 180d⟩+≡ (55a) ◁180c
```
else if(strcmp(cb->f[0], "remmulti") == 0){
    if(cb->nf < 2)
        error("remmulti needs interface address");
    if(!ipismulticast(cv->raddr))
        error("remmulti for a non multicast address");
    if (parseip(ia, cb->f[1]) == -1)
        error(Ebadip);
    ipifcremmulti(cv, cv->raddr, ia);
}
```
Uses ipifcremmulti() 382b, ipismulticast() 381c, and parseip() 89.

⟨ipifcctl() *else if other string* 180e⟩+≡ (71a) ◁105c 180f▷
```
else if(strcmp(argv[0], "joinmulti") == 0)
    return ipifcjoinmulti(ifc, argv, argc);
```
Uses ipifcjoinmulti() 383a.

⟨ipifcctl() *else if other string* 180f⟩+≡ (71a) ◁180e 517a▷
```
else if(strcmp(argv[0], "leavemulti") == 0)
    return ipifcleavemulti(ifc, argv, argc);
```
Uses ipifcleavemulti() 383b.

⟨Medium(kernel) *multicast methods* 180g⟩≡ (26c)
```
/* for arming interfaces to receive multicast */
void  (*addmulti)(Ipifc *ifc, uchar *a, uchar *ia);
void  (*remmulti)(Ipifc *ifc, uchar *a, uchar *ia);

/* for routing multicast groups */
void  (*joinmulti)(Ipifc *ifc, uchar *a, uchar *ia);
void  (*leavemulti)(Ipifc *ifc, uchar *a, uchar *ia);
```

⟨Conv(kernel) *multicast fields* 181a⟩≡ (33e)
```
  Ipmulti *multi;     /* multicast bindings for this interface */
```

⟨closeconv() *locals* 181b⟩≡ (50c)
```
  Ipmulti *mp;
```

⟨closeconv() *if multi, call ipifcremmulti* 181c⟩≡ (50c)
```
  while((mp = cv->multi) != nil)
      ipifcremmulti(cv, mp->ma, mp->ia);
```
Uses ipifcremmulti() 382b.

⟨Netif(kernel) *multicast fields* 181d⟩≡ (36a)
```
  Netaddr *maddr;     /* known multicast addresses */
  int nmaddr;     /* number of known multicast addresses */
  Netaddr *mhash[Nmhash];   /* hash table of multicast addresses */
```
Uses Nmhash 240a.

⟨Netif(kernel) *multicast methods* 181e⟩≡ (36a)
```
  void  (*multicast)(void*, uchar*, int);
```

21.5 Network database: ndb

/net/ndb exposes the network database—a simple text format that maps hostnames to IP addresses, DNS servers, gateways, and other configuration. Programs like ipconfig write entries (e.g., sys=myhost ip=10.0.2.15) after DHCP, and the connection server reads them for name resolution. The 1024-byte buffer in Fs is small because it only holds the local machine's configuration.

⟨Fs(kernel) *ndb fields* 181f⟩≡ (29d)
```
  char  ndb[1024];    /* an ndb entry for this interface */
  int ndbvers;
  long  ndbmtime;
```

21.5.1 /net/ndb

⟨Qid *toplevel extra cases* 181g⟩+≡ (34i) ◁130a 189d▷
```
  Qndb,
```

⟨ip1gen() *switch TYPE qid cases* 181h⟩+≡ (46c) ◁130b 189e▷
```
  case Qndb:
      p = "ndb";
      len = strlen(f->ndb);
      q.vers = f->ndbvers;
      break;
```
Uses Qndb-220 181g.

⟨ip1gen() *locals* 181i⟩≡ (46c)
```
  extern ulong    kerndate;
```

⟨ipgen() *if Qndb, adjust mtime* 181j⟩≡ (46c)
```
  if(i == Qndb && f->ndbmtime > kerndate)
      dp->mtime = f->ndbmtime;
```
Uses Qndb-220 181g.

⟨ipopen() *switch TYPE qid cases* 182a⟩+≡ (49a) ◁108b 189f▷
```
  case Qndb:
      if(omode & (OWRITE|OTRUNC) && !iseve())
          error(Eperm);
      if((omode & (OWRITE|OTRUNC)) == (OWRITE|OTRUNC))
          f->ndb[0] = 0;
      break;
```
Uses Qndb-220 181g.

⟨ipread() *switch TYPE qid cases* 182b⟩+≡ (51a) ◁130c 189h▷
```
  case Qndb:
      return readstr(offset, a, n, f->ndb);
```
Uses Qndb-220 181g and readstr().

⟨ipwrite() *switch TYPE qid cases* 182c⟩+≡ (51c) ◁120c 189i▷
```
  case Qndb:
      return ndbwrite(f, a, offset, n);
      break;
```
Uses Qndb-220 181g and ndbwrite() 331b.

21.6 Connection Server

21.7 Sniffing

21.8 Point to point interface

⟨ipifcadd() *check for point to point interface* 182d⟩≡ (71c)
```
  /* check for point-to-point interface */
  if(ipcmp(ip, v6loopback)) /* skip v6 loopback, it's a special address */
  if(ipcmp(mask, IPallbits) == 0)
      type |= Rptpt;
```
Uses IPallbits 227d, Rptpt 38e, ipcmp 23d, and v6loopback 371b.

⟨ipifcadd() *register proxy if point to point interface or proxy* 182e⟩≡ (71c)
```
  if((type & (Rproxy|Rptpt)) == (Rproxy|Rptpt)){
      ipifcregisterproxy(f, ifc, rem);
      goto out;
  }
```
Uses Rproxy 38e, Rptpt 38e, and ipifcregisterproxy() 383c.

21.9 Proxy

⟨ipifcadd() *switch argc, proxy case, and fall through* 182f⟩≡ (71c)
```
  // add <ip> <mask> <rem> <mtu> proxy
  case 6:
      if(strcmp(argv[5], "proxy") == 0)
          type |= Rproxy;
      /* fall through */
```
Uses Rproxy 38e.

21.10 Wifi

21.11 VLAN

21.12 VPN

21.13 Packet filter

Chapter 22

Conclusion

You now know how the Plan 9 network stack works, to the smallest details, and more generally how many network stacks work.

The Plan 9 network stack is implemented entirely in the kernel as a set of protocol modules behind the `/net/` file interface. Each protocol (Ethernet, IP, ARP, UDP, IL, TCP) registers a `Proto` structure with connect, announce, receive, and close methods—the same device-polymorphism pattern used throughout the Plan 9 kernel. A network connection is a directory under `/net/<proto>/<conv>/` with `data`, `ctl`, `status`, and other files, so programs interact with the network by reading and writing files rather than through a special socket API. Along the way, you have seen how the stack is layered: Ethernet frames carry IP datagrams, IP datagrams carry UDP or IL or TCP segments, and each layer demultiplexes incoming packets to the right conversation based on addresses and port numbers. You have also seen the fundamental problem of reliable communication—how IL (and TCP) provide ordered, reliable byte streams over an unreliable packet network using sequence numbers, acknowledgments, retransmission timers, and flow control.

22.1 Patterns and techniques

These techniques apply far beyond protocol stacks:

- *Protocol layering*: each layer wraps the one above in its own header; incoming data is unwrapped layer by layer. The same onion-skin structure appears in web middleware stacks (each layer adding authentication, logging, compression) and compiler passes (each transforming the IR).
- *Multiplexing*: sharing one resource among many logical channels via a dispatch table keyed by identifier. HTTP/2 multiplexes streams over one TCP connection, `tmux` multiplexes terminals over one SSH session, and event loops multiplex file descriptors through `select/epoll`.
- *Reliable delivery over unreliable channels*: sequence numbers, acknowledgments, and retransmission are not specific to networking—message queues (Kafka, RabbitMQ) and distributed databases use the same mechanisms to cope with unreliable communication.
- *Plugin architecture*: each protocol registers a `Proto` structure with method pointers; adding a new protocol requires no changes to existing code. The same extensibility pattern is used by web server modules (Apache, Nginx) and database storage engines (MySQL's pluggable engines).

22.2 Connections to other books

- Kernel book [Pad14]: the network stack runs inside the kernel and uses its facilities: `Dev` for the `/net/` device, `sleep/wakeup` for blocking on connections, `kproc` for protocol processing threads, and the `Ipifc` structure to manage network interfaces.

- Libcore book [Pad16]: the user-space `dial()` function in `libc` opens connections through `/net/`. The chapter on networking in Libcore book [Pad16] covers the user-facing API; this book covers what happens inside the kernel when `dial()` writes to a `ctl` file.

22.3 Beyond the Plan 9 network stack

The Plan 9 network stack implements the core protocols cleanly but is small compared to modern networking subsystems. Here are some areas where they differ:

- *TCP*: the Plan 9 TCP implementation is functional but basic. Modern TCP implementations (Linux, FreeBSD) include decades of refinements: selective acknowledgment (SACK), window scaling, congestion control algorithms (Cubic, BBR), TCP Fast Open, and Explicit Congestion Notification (ECN). These optimizations make a large difference at scale—BBR alone can double throughput on lossy links.

- *Congestion control*: Plan 9's IL protocol uses a simple retransmission timer. Modern congestion control is a rich field: algorithms like BBR model the network's bandwidth and round-trip time to avoid both underutilization and congestion collapse. This is one of the most actively researched areas in networking.

- *Zero-copy and kernel bypass*: for high-performance networking, Linux provides `sendfile()`, `splice()`, and `io_uring` to avoid copying data between kernel and user space. DPDK and RDMA bypass the kernel entirely, giving user-space programs direct access to the NIC for microsecond-level latency.

- *Programmable networking*: Linux's eBPF allows inserting custom programs at various points in the network stack—packet filtering, load balancing, traffic shaping—without modifying kernel code. XDP (eXpress Data Path) processes packets before they enter the stack, enabling line-rate packet processing.

- *TLS and QUIC*: TLS (Transport Layer Security) is now expected on nearly all connections. QUIC (HTTP/3) combines transport and encryption into a single UDP-based protocol with faster connection setup and better performance on lossy networks. The Plan 9 network stack has basic SSL support but nothing comparable.

- *IPv6*: the Plan 9 stack includes some IPv6 support, but modern stacks must handle dual-stack operation, IPv6 extension headers, neighbor discovery, and the transition mechanisms needed for a world that is still migrating from IPv4.

The fundamentals are the same everywhere: layered protocols, packet demultiplexing, reliable delivery through acknowledgments and retransmission. The Plan 9 network stack presents these ideas clearly, with the added elegance of exposing everything through the file system. Modern stacks are larger because they optimize for datacenter-scale throughput and a security landscape that did not exist when Plan 9 was designed.

Appendix A

Debugging

Network debugging relies heavily on formatted output of addresses and headers. The `eipfmt` function registers format verbs for `print`: `%E` for Ethernet MAC addresses (like `00:11:22:33:44:55`), `%I` for IP addresses (like `10.0.0.2` or IPv6 with `::` elision), and `%M` for subnet masks.

A.1 Dumpers

A.1.1 Addresses

⟨*function* `eipfmt` 186⟩≡ (226e)

```
int
eipfmt(Fmt *f)
{
    char buf[5*8];
    static char *efmt = "%.2ux%.2ux%.2ux%.2ux%.2ux%.2ux"; // ethernet
    static char *ifmt = "%d.%d.%d.%d"; // internet, v4
    uchar *p;
    ipaddr ip;
    ulong *lp;
    ushort s;
    int i, j, n, eln, eli;

    switch(f->r) {
    case 'E':  /* Ethernet address */
        p = va_arg(f->args, uchar*);
        snprint(buf, sizeof buf, efmt, p[0], p[1], p[2], p[3], p[4], p[5]);
        return fmtstrcpy(f, buf);

    case 'I':  /* Ip address */
        p = va_arg(f->args, uchar*);
common:
        if(memcmp(p, v4prefix, 12) == 0){
            snprint(buf, sizeof buf, ifmt, p[12], p[13], p[14], p[15]);
            return fmtstrcpy(f, buf);
        }

        /* find longest elision */
        eln = eli = -1;
        for(i = 0; i < 16; i += 2){
            for(j = i; j < 16; j += 2)
                if(p[j] != 0 || p[j+1] != 0)
                    break;
            if(j > i && j - i > eln){
```

```
                eli = i;
                eln = j - i;
            }
        }

        /* print with possible elision */
        n = 0;
        for(i = 0; i < 16; i += 2){
            if(i == eli){
                n += sprint(buf+n, "::");
                i += eln;
                if(i >= 16)
                    break;
            } else if(i != 0)
                n += sprint(buf+n, ":");
            s = (p[i]<<8) + p[i+1];
            n += sprint(buf+n, "%ux", s);
        }
        return fmtstrcpy(f, buf);

    case 'i':  /* v6 address as 4 longs */
        lp = va_arg(f->args, ulong*);
        for(i = 0; i < 4; i++)
            hnputl(ip+4*i, *lp++);
        p = ip;
        goto common;

    case 'V':  /* v4 ip address */
        p = va_arg(f->args, uchar*);
        snprint(buf, sizeof buf, ifmt, p[0], p[1], p[2], p[3]);
        return fmtstrcpy(f, buf);

    case 'M':  /* ip mask */
        p = va_arg(f->args, uchar*);

        /* look for a prefix mask */
        for(i = 0; i < 16; i++)
            if(p[i] != 0xff)
                break;
        if(i < 16){
            if((prefixvals[p[i]] & Isprefix) == 0)
                goto common;
            for(j = i+1; j < 16; j++)
                if(p[j] != 0)
                    goto common;
            n = 8*i + (prefixvals[p[i]] & ~Isprefix);
        } else
            n = 8*16;

        /* got one, use /xx format */
        snprint(buf, sizeof buf, "/%d", n);
        return fmtstrcpy(f, buf);
    }
    return fmtstrcpy(f, "(eipfmt)");
}
```

Uses Isprefix-3 226c, prefixvals 226d, and v4prefix 20e.

A.1.2 IP interface

⟨*function* `main` 188a⟩≡ (231c)

```
void
main(void)
{
    Ipifc *ifc, *list;
    Iplifc *lifc;
    int i;

    fmtinstall('I', eipfmt);
    fmtinstall('M', eipfmt);

    list = readipifc("/net", nil, -1);
    for(ifc = list; ifc; ifc = ifc->next){
        print("ipifc %s %d\n", ifc->dev, ifc->mtu);
        for(lifc = ifc->lifc; lifc; lifc = lifc->next)
            print("\t%I %M %I\n", lifc->ip, lifc->mask, lifc->net);
    }
}
```

A.2 `/net/log`

`/net/log` provides a kernel-level network event log. The `Netlog` structure holds a circular buffer where `netlog()` calls write formatted messages. The `logmask` controls which protocol layers produce output (each protocol has a bit: `Logip`, `Logudp`, `Logil`, etc.), and `iponly` can filter messages for a single IP address. This is the primary debugging tool for kernel network code—it avoids the overhead of `print` and works even when the network itself is broken.

A.2.1 `Netlog`

⟨*struct* `Netlog` 188b⟩≡ (389b)

```
/*
 *  action log
 */
struct Netlog {
    int opens;

    // array<char> of size Nlog
    char*   buf;
    int len;

    char    *rptr;
    char    *end;


    int logmask;            /* mask of things to debug */
    ipaddr   iponly;      /* ip address to print debugging for */
    int iponlyset;

    // Extra
    Lock;
    QLock;
    Rendez;
};
```

⟨*enum* _anon_ (kernel/network/ip/netlog.c) 189a⟩≡ (389b)
```
enum {
    Nlog        = 16*1024,
};
```

⟨Fs(kernel) *logging fields* 189b⟩≡ (29d)
```
Netlog  *alog;
```

⟨*function* netloginit 189c⟩≡ (389b)
```
void
netloginit(Fs *f)
{
    f->alog = smalloc(sizeof(Netlog));
}
```

A.2.2 /net/log

⟨Qid *toplevel extra cases* 189d⟩+≡ (34i) ◁181g
```
Qlog,
```

⟨ip1gen() *switch TYPE qid cases* 189e⟩+≡ (46c) ◁181h
```
case Qlog:
    p = "log";
    break;
```
Uses Qlog-221 189d.

⟨ipopen() *switch TYPE qid cases* 189f⟩+≡ (49a) ◁182a 193b▷
```
case Qlog:
    netlogopen(f);
    break;
```
Uses Qlog-221 189d and netlogopen() 190a.

⟨ipclose() *switch TYPE qid cases* 189g⟩+≡ (50a) ◁50b 193c▷
```
case Qlog:
    if(c->flag & COPEN)
        netlogclose(f);
    break;
```
Uses Qlog-221 189d and netlogclose() 388d.

⟨ipread() *switch TYPE qid cases* 189h⟩+≡ (51a) ◁182b 193d▷
```
case Qlog:
    return netlogread(f, a, offset, n);
```
Uses Qlog-221 189d and netlogread() 190b.

⟨ipwrite() *switch TYPE qid cases* 189i⟩+≡ (51c) ◁182c
```
case Qlog:
    netlogctl(f, a, n);
    return n;
```
Uses Qlog-221 189d and netlogctl() 191d.

A.2.3 Opening

⟨*function* `netlogopen` 190a⟩≡ (389b)

```
void
netlogopen(Fs *f)
{
    lock(f->alog);
    if(waserror()){
        unlock(f->alog);
        nexterror();
    }
    if(f->alog->opens == 0){
        if(f->alog->buf == nil)
            f->alog->buf = malloc(Nlog);
        if(f->alog->buf == nil)
            error(Enomem);
        f->alog->rptr = f->alog->buf;
        f->alog->end = f->alog->buf + Nlog;
    }
    f->alog->opens++;
    unlock(f->alog);
    poperror();
}
```

Uses `Nlog-253` 189a.

A.2.4 Reading

⟨*function* `netlogread` 190b⟩≡ (389b)

```
long
netlogread(Fs *f, void *a, ulong, long n)
{
    int i, d;
    char *p, *rptr;

    qlock(f->alog);
    if(waserror()){
        qunlock(f->alog);
        nexterror();
    }

    for(;;){
        lock(f->alog);
        if(f->alog->len){
            if(n > f->alog->len)
                n = f->alog->len;
            d = 0;
            rptr = f->alog->rptr;
            f->alog->rptr += n;
            if(f->alog->rptr >= f->alog->end){
                d = f->alog->rptr - f->alog->end;
                f->alog->rptr = f->alog->buf + d;
            }
            f->alog->len -= n;
            unlock(f->alog);

            i = n-d;
            p = a;
            memmove(p, rptr, i);
            memmove(p+i, f->alog->buf, d);
```

```
            break;
        }
        else
            unlock(f->alog);

        sleep(f->alog, netlogready, f);
    }

    qunlock(f->alog);
    poperror();

    return n;
}
```

Uses `netlogready()` 191a.

⟨*function* `netlogready` 191a⟩≡ (389b)

```
static bool
netlogready(void *a)
{
    Fs *f = a;

    return f->alog->len;
}
```

A.2.5 Control

⟨*enum* `_anon_ (kernel/network/ip/netlog.c)2` 191b⟩≡ (389b)

```
enum
{
    CMset,
    CMclear,
    CMonly,
};
```

⟨*global* `routecmd` 191c⟩≡ (389b)

```
static
Cmdtab routecmd[] = {
    CMset,      "set",      0,
    CMclear,    "clear",    0,
    CMonly,     "only",     0,
};
```

Uses `CMclear-256` 191b, `CMonly-257` 191b, and `CMset-255` 191b.

⟨*function* `netlogctl` 191d⟩≡ (389b)

```
void
netlogctl(Fs *f, char* s, int n)
{
    int i, set;
    Netlogflag *fp;
    Cmdbuf *cb;
    Cmdtab *ct;

    cb = parsecmd(s, n);
    if(waserror()){
        free(cb);
        nexterror();
    }

    if(cb->nf < 2)
```

```
        error(Ebadnetctl);

    ct = lookupcmd(cb, routecmd, nelem(routecmd));

    SET(set);

    switch(ct->index){
    case CMset:
        set = 1;
        break;

    case CMclear:
        set = 0;
        break;

    case CMonly:
        parseip(f->alog->iponly, cb->f[1]);
        if(ipcmp(f->alog->iponly, IPnoaddr) == 0)
            f->alog->iponlyset = 0;
        else
            f->alog->iponlyset = 1;
        free(cb);
        poperror();
        return;

    default:
        cmderror(cb, "unknown netlog control message");
    }

    for(i = 1; i < cb->nf; i++){
        for(fp = flags; fp->name; fp++)
            if(strcmp(fp->name, cb->f[i]) == 0)
                break;
        if(fp->name == nil)
            continue;
        if(set)
            f->alog->logmask |= fp->mask;
        else
            f->alog->logmask &= ~fp->mask;
    }

    free(cb);
    poperror();
}
```

Uses `CMclear-256` 191b, `CMonly-257` 191b, `CMset-255` 191b, `Ebadnetctl` 388c, `IPnoaddr` 23b, `flags-254` 388b, `ipcmp` 23d, `lookupcmd()`, `parsecmd()`, `parseip()` 89, and `routecmd-258` 191c.

A.3 /bin/snoopy

A.3.1 /net/ipifc/x/snoop

⟨`Qid` *conversation extra cases, last entry* 192a⟩≡ (34i)

```
Qsnoop,
```

⟨`Conv(kernel)` *snoop fields* 192b⟩≡ (33e)

```
Ref snoopers;    /* number of processes with snoop open */
Queue*  sq;      /* snooping queue */
```

⟨ip3gen() *switch TYPE qid cases* 193a⟩+≡ (48a) ◁65a

```
  case Qsnoop:
      if(strcmp(cv->p->name, "ipifc") != 0)
          return -1;
      devdir(c, q, "snoop", qlen(cv->sq), cv->owner, 0400, dp);
      return 1;
```

Uses `Qsnoop-235` 192a.

⟨ipopen() *switch TYPE qid cases* 193b⟩+≡ (49a) ◁189f

```
  case Qsnoop:
      if(omode != OREAD)
          error(Eperm);
      p = f->p[PROTO(c->qid)];
      cv = p->conv[CONV(c->qid)];
      if(strcmp(ATTACHER(c), cv->owner) != 0 && !iseve())
          error(Eperm);
      incref(&cv->snoopers);
      break;
```

Uses `ATTACHER-250` 41b, `CONV-246` 35d, `PROTO-247` 35c, and `Qsnoop-235` 192a.

⟨ipclose() *switch TYPE qid cases* 193c⟩+≡ (50a) ◁189g

```
  case Qsnoop:
      if(c->flag & COPEN)
          decref(&f->p[PROTO(c->qid)]->conv[CONV(c->qid)]->snoopers);
      break;
```

Uses `CONV-246` 35d, `PROTO-247` 35c, and `Qsnoop-235` 192a.

⟨ipread() *switch TYPE qid cases* 193d⟩+≡ (51a) ◁189h

```
  case Qsnoop:
      cv = f->p[PROTO(ch->qid)]->conv[CONV(ch->qid)];
      return qread(cv->sq, a, n);
```

Uses `CONV-246` 35d, `PROTO-247` 35c, and `Qsnoop-235` 192a.

A.3.2 /bin/snoopy

Appendix B

Profiling

B.1 `/net/x/stats`

Appendix C

Error Management

Appendix D

Ethernet NE2000 Driver

The NE2000 is a classic Ethernet card based on the National Semiconductor DP8390 chip—one of the simplest Ethernet controllers, which makes it ideal for learning. QEMU emulates it, so this is the driver used by the Principia Softwarica system. The driver manages a ring buffer of pages on the card for receiving packets, and uses programmed I/O (port reads/writes) to transfer data between the card and system memory.

D.1 Data structures

The `Dp8390` control block maps the card's hardware state: the I/O port address, the ring buffer boundaries (`pstart` to `pstop`), the next expected receive packet (`nxtpkt`), and the transmit busy flag. The ring buffer is a circular region of on-card memory divided into 256-byte pages; the DP8390 chip writes received packets into consecutive pages and wraps around at `pstop`.

D.1.1 Ethernet `Dp8390` controller

⟨*struct* Dp8390 196⟩≡ (266a)

```
/*
 * Ctlr for the boards using the National Semiconductor DP8390
 * and SMC 83C90 Network Interface Controller.
 * Common code is in ether8390.c.
 */
struct Dp8390 {
    ulong port;   /* I/O address of 8390 */
    ulong data;   /* I/O data port if no shared memory */

    uchar width;   /* data transfer width in bytes */
    bool ram;   /* true if card has shared memory */
    uchar dummyrr;  /* do dummy remote read */

    uchar nxtpkt;   /* receive: software bndry */
    uchar pstart;
    uchar pstop;

    int txbusy;   /* transmit */
    uchar tstart;   /* 8390 ring addresses */

    ⟨Dp8390 multicast fields 210a⟩

    // Extra
    Lock;
};
```

D.2 Initialisation

NE2000 initialization proceeds in three phases: **ether2000link** registers the driver with the Ethernet subsystem, **ne2000reset** probes the hardware (reading the MAC address from the card's PROM, configuring the ring buffer layout), and **dp8390attach** starts the interrupt handler. The ring buffer is partitioned: the first page is reserved for transmit, the rest is the receive ring.

⟨*function* `ether2000link` 197a⟩≡ (270b)

```
void
ether2000link(void)
{
    addethercard("NE2000", ne2000reset);
}
```

Uses `ne2000reset()` 197b.

⟨*function* `ne2000reset` 197b⟩≡ (270b)

```
static errorneg1
ne2000reset(Ether* edev)
{
    ushort buf[16];
    ulong port;
    Dp8390 *dp8390;
    int i;
    eaddr ea;

    if(edev->port == 0)
        ne2000pnp(edev);
    if(edev->port == 0)
        return ERROR_NEG1;

    /*
     * Set up the software configuration.
     * Use defaults for irq, mem and size
     * if not specified.
     * Must have a port, no more default.
     */
    if(edev->irq == 0)
        edev->irq = 2;
    if(edev->mem == 0)
        edev->mem = 0x4000;
    if(edev->size == 0)
        edev->size = 16*1024;
    port = edev->port;

    if(ioalloc(edev->port, 0x20, 0, "ne2000") < 0)
        return ERROR_NEG1;

    edev->ctlr = malloc(sizeof(Dp8390));
    dp8390 = edev->ctlr;
    if(dp8390 == nil)
        error(Enomem);

    dp8390->width = 2;
    dp8390->ram = false;

    dp8390->port = port;
    dp8390->data = port+Data;

    dp8390->tstart = HOWMANY(edev->mem, Dp8390BufSz);
```

```
    dp8390->pstart = dp8390->tstart + HOWMANY(sizeof(Etherpkt), Dp8390BufSz);
    dp8390->pstop = dp8390->tstart + HOWMANY(edev->size, Dp8390BufSz);

    dp8390->dummyrr = 1;
    for(i = 0; i < edev->nopt; i++){
        if(strcmp(edev->opt[i], "nodummyrr"))
            continue;
        dp8390->dummyrr = 0;
        break;
    }

    /*
     * Reset the board. This is done by doing a read
     * followed by a write to the Reset address.
     */
    buf[0] = inb(port+Reset);
    arch_delay(2);
    outb(port+Reset, buf[0]);
    arch_delay(2);

    /*
     * Init the (possible) chip, then use the (possible)
     * chip to read the (possible) PROM for ethernet address
     * and a marker byte.
     * Could just look at the DP8390 command register after
     * initialisation has been tried, but that wouldn't be
     * enough, there are other ethernet boards which could
     * match.
     * Parallels has buf[0x0E] == 0x00 whereas real hardware
     * usually has 0x57.
     */

    // will fill in the Ether callbacks
    dp8390reset(edev);

    memset(buf, 0, sizeof(buf));
    dp8390read(dp8390, buf, 0, sizeof(buf));
    i = buf[0x0E] & 0xFF;
    if((i != 0x00 && i != 0x57) || (buf[0x0F] & 0xFF) != 0x57){
        iofree(edev->port);
        free(edev->ctlr);
        return ERROR_NEG1;
    }

    /*
     * Stupid machine. Shorts were asked for,
     * shorts were delivered, although the PROM is a byte array.
     * Set the ethernet address.
     */
    memset(ea, 0, Eaddrlen);
    if(memcmp(ea, edev->ea, Eaddrlen) == 0){
        for(i = 0; i < sizeof(edev->ea); i++)
            edev->ea[i] = buf[i];
    }
    dp8390setea(edev);

    return OK_0;
}
```

Uses `Data-376` 270a, `Dp8390BufSz` 265c, `Eaddrlen` 27d, `Reset-377` 270a, `dp8390read()` 199a, `dp8390reset()` 199b,

dp8390setea() 200b, and ne2000pnp() 212e.

⟨*function* dp8390read 199a⟩≡ (269)

```
void*
dp8390read(Dp8390* ctlr, void* to, ulong from, ulong len)
{
	void *v;

	ilock(ctlr);
	v = _dp8390read(ctlr, to, from, len);
	iunlock(ctlr);

	return v;
}
```

Uses _dp8390read() 208b.

⟨*function* dp8390reset 199b⟩≡ (269)

```
int
dp8390reset(Ether* ether)
{
	Dp8390 *ctlr;

	ctlr = ether->ctlr;

	/*
	 * This is the initialisation procedure described
	 * as 'mandatory' in the datasheet, with references
	 * to the 3C503 technical reference manual.
	 */
	disable(ctlr);
	if(ctlr->width != 1)
		regw(ctlr, Dcr, Ft4WORD|Ls|Wts);
	else
		regw(ctlr, Dcr, Ft4WORD|Ls);

	regw(ctlr, Rbcr0, 0);
	regw(ctlr, Rbcr1, 0);

	regw(ctlr, Tcr, LpbkNIC);
	regw(ctlr, Rcr, Mon);

	/*
	 * Init the ring hardware and software ring pointers.
	 * Can't initialise ethernet address as it may not be
	 * known yet.
	 */
	ringinit(ctlr);
	regw(ctlr, Tpsr, ctlr->tstart);

	/*
	 * Clear any pending interrupts and mask then all off.
	 */
	regw(ctlr, Isr, 0xFF);
	regw(ctlr, Imr, 0);

	/*
	 * Leave the chip initialised,
	 * but in monitor mode.
	 */
	regw(ctlr, Cr, Page0|RdABORT|Sta);
```

```
    /*
     * Set up the software configuration.
     */
    ether->attach = attach;
    ether->shutdown = shutdown;

    ether->transmit = transmit;
    ether->interrupt = interrupt;
    ether->ifstat = 0;

    ether->promiscuous = promiscuous;
    ether->multicast = multicast;
    ether->arg = ether;

    return 0;
}
```

Uses Cr-381 266b, Dcr-406 266b, Ft4WORD-443 267c, Imr-407 266b, Isr-388 266b, LpbkNIC-449 268a, Ls-437 267c, Mon-466 268c, Page0-423 267a, Rbcr0-402 266b, Rbcr1-403 266b, Rcr-404 266b, RdABORT-420 267a, Sta-412 267a, Tcr-405 266b, Tpsr-397 266b, Wts-434 267c, attach() 201c, disable() 201a, interrupt() 205b, multicast() 210c, promiscuous() 212a, regw 202a, ringinit() 200a, shutdown() 201b, and transmit() 202c.

⟨*function* ringinit 200a⟩≡ (269)

```
static void
ringinit(Dp8390* ctlr)
{
    regw(ctlr, Pstart, ctlr->pstart);
    regw(ctlr, Pstop, ctlr->pstop);
    regw(ctlr, Bnry, ctlr->pstop-1);

    regw(ctlr, Cr, Page1|RdABORT|Stp);
    regw(ctlr, Curr, ctlr->pstart);
    regw(ctlr, Cr, Page0|RdABORT|Stp);

    ctlr->nxtpkt = ctlr->pstart;
}
```

Uses Bnry-384 266b, Cr-381 266b, Curr-409 266b, Page0-423 267a, Page1-424 267a, Pstart-395 266b, Pstop-396 266b, RdABORT-420 267a, Stp-411 267a, and regw 202a.

⟨*function* dp8390setea 200b⟩≡ (269)

```
void
dp8390setea(Ether* ether)
{
    int i;
    uchar cr;
    Dp8390 *ctlr;

    ctlr = ether->ctlr;

    /*
     * Set the ethernet address into the chip.
     * Take care to restore the command register
     * afterwards. Don't care about multicast
     * addresses as multicast is never enabled
     * (currently).
     */
    ilock(ctlr);
    cr = regr(ctlr, Cr) & ~Txp;
```

```
    regw(ctlr, Cr, Page1|(~(Ps1|Ps0) & cr));
    for(i = 0; i < Eaddrlen; i++)
        regw(ctlr, Par0+i, ether->ea[i]);
    regw(ctlr, Cr, cr);
    iunlock(ctlr);
}
```

Uses Cr-381 266b, Eaddrlen 27d, Page1-424 267a, Par0-408 266b, Ps0-421 267a, Ps1-422 267a, Txp-413 267a, regr 202b, and regw 202a.

⟨*function* disable 201a⟩≡ (269)

```
static void
disable(Dp8390* ctlr)
{
    int timo;

    /*
     * Stop the chip. Set the Stp bit and wait for the chip
     * to finish whatever was on its tiny mind before it sets
     * the Rst bit.
     * The timeout is needed because there may not be a real
     * chip there if this is called when probing for a device
     * at boot.
     */
    regw(ctlr, Cr, Page0|RdABORT|Stp);
    regw(ctlr, Rbcr0, 0);
    regw(ctlr, Rbcr1, 0);
    for(timo = 10000; (regr(ctlr, Isr) & Rst) == 0 && timo; timo--)
            ;
}
```

Uses Cr-381 266b, Isr-388 266b, Page0-423 267a, Rbcr0-402 266b, Rbcr1-403 266b, RdABORT-420 267a, Rst-433 267b, Stp-411 267a, regr 202b, and regw 202a.

⟨*function* shutdown 201b⟩≡ (269)

```
static void
shutdown(Ether *ether)
{
    Dp8390 *ctlr;

    ctlr = ether->ctlr;
    disable(ctlr);
}
```

Uses disable() 201a.

D.3 Mounting

`attach` takes the card out of monitor mode and enables transmit/receive. It clears the missed-packet counter, acknowledges any pending interrupts, and sets the interrupt mask to listen for packet-received, transmit-complete, and error conditions. After this, the card is live on the network.

⟨*function* attach 201c⟩≡ (269)

```
static void
attach(Ether* ether)
{
    Dp8390 *ctlr;
    uchar r;

    ctlr = ether->ctlr;
```

```
    /*
     * Enable the chip for transmit/receive.
     * The init routine leaves the chip in monitor
     * mode. Clear the missed-packet counter, it
     * increments while in monitor mode.
     * Sometimes there's an interrupt pending at this
     * point but there's nothing in the Isr, so
     * any pending interrupts are cleared and the
     * mask of acceptable interrupts is enabled here.
     */
    r = Ab;
    if(ether->prom)
        r |= Pro;
    if(ether->nmaddr)
        r |= Am;
    ilock(ctlr);
    regw(ctlr, Isr, 0xFF);
    regw(ctlr, Imr, Cnt|Ovw|Txe|Rxe|Ptx|Prx);
    regw(ctlr, Rcr, r);
    r = regr(ctlr, Ref2);
    regw(ctlr, Tcr, LpbkNORMAL);
    iunlock(ctlr);
    USED(r);
}
```

Uses `Ab-463` 268c, `Am-464` 268c, `Cnt-431` 267b, `Imr-407` 266b, `Isr-388` 266b, `LpbkNORMAL-448` 268a, `Ovw-430` 267b, `Pro-465` 268c, `Prx-426` 267b, `Ptx-427` 267b, `Rcr-404` 266b, `Ref2-394` 266b, `Rxe-428` 267b, `Tcr-405` 266b, `Txe-429` 267b, `regr` 202b, and `regw` 202a.

D.4 IO

The NE2000 uses programmed I/O: all communication with the card happens through x86 `inb`/`outb` port instructions. `regr` and `regw` read and write DP8390 registers; data transfer uses DMA-like "remote read" and "remote write" commands that move data between host memory and the card's ring buffer one word at a time. Writing transmits a packet by copying it to the transmit page and issuing a transmit command; reading walks the receive ring, extracting packets from consecutive pages.

⟨*macro* `regw` 202a⟩≡ (266a)

```
#define regw(c, r, v) outb((c)->port+(r), (v))
```

⟨*macro* `regr` 202b⟩≡ (266a)

```
/*
 * x86-specific code.
 */
#define regr(c, r) inb((c)->port+(r))
```

D.4.1 Writing

Transmitting a packet on the NE2000 involves three steps: dequeue a block from the output queue, copy it to the card's transmit buffer using the DP8390's remote-DMA write command (transferring 2 bytes at a time via `outss`), and issue a transmit command. The card handles the rest: preamble, CRC, and carrier sense.

⟨*function* `transmit` *(*`(kernel/network/386/ether8390.c)`*)* 202c⟩≡ (269)

```
static void
transmit(Ether* ether)
{
```

```
    Dp8390 *ctlr;

    ctlr = ether->ctlr;

    ilock(ctlr);
    txstart(ether);
    iunlock(ctlr);
}
```

Uses `txstart()` 203.

⟨*function* `txstart` 203⟩≡ (269)

```
static void
txstart(Ether* ether)
{
    int len;
    Dp8390 *ctlr;
    Block *bp;
    uchar minpkt[ETHERMINTU], *rp;

    ctlr = ether->ctlr;

    /*
     * This routine is called both from the top level and from interrupt
     * level and expects to be called with ctlr already locked.
     */
    if(ctlr->txbusy)
        return;
    bp = qget(ether->oq);
    if(bp == nil)
        return;

    /*
     * Make sure the packet is of minimum length;
     * copy it to the card's memory by the appropriate means;
     * start the transmission.
     */
    len = BLEN(bp);
    rp = bp->rp;
    if(len < ETHERMINTU){
        rp = minpkt;
        memmove(rp, bp->rp, len);
        memset(rp+len, 0, ETHERMINTU-len);
        len = ETHERMINTU;
    }

    if(ctlr->ram)
        memmove((void*)(ether->mem+ctlr->tstart*Dp8390BufSz), rp, len);
    else
        dp8390write(ctlr, ctlr->tstart*Dp8390BufSz, rp, len);
    freeb(bp);

    regw(ctlr, Tbcr0, len & 0xFF);
    regw(ctlr, Tbcr1, (len>>8) & 0xFF);
    regw(ctlr, Cr, Page0|RdABORT|Txp|Sta);

    ether->outpackets++;
    ctlr->txbusy = 1;
}
```

Uses `Cr-381` 266b, `Dp8390BufSz` 265c, `ETHERMINTU` 240f, `Page0-423` 267a, `RdABORT-420` 267a, `Sta-412` 267a, `Tbcr0-398` 266b, `Tbcr1-399` 266b, `Txp-413` 267a, `dp8390write()` 204, and `regw` 202a.

⟨*function* dp8390write 204⟩≡ (269)

```
static void*
dp8390write(Dp8390* ctlr, ulong to, void* from, ulong len)
{
    ulong crda;
    uchar cr;
    int timo, width;

top:
    /*
     * Write some data to offset 'to' in the card's memory
     * using the DP8390 remote DMA facility, reading it at
     * 'from' in main memory, via the I/O data port.
     */
    cr = regr(ctlr, Cr) & ~Txp;
    regw(ctlr, Cr, Page0|RdABORT|Sta);
    regw(ctlr, Isr, Rdc);

    len = ROUNDUP(len, ctlr->width);

    /*
     * Set up the remote DMA address and count.
     * This is straight from the DP8390[12D] datasheet,
     * hence the initial set up for read.
     * Assumption here that the A7000 EtherV card will
     * never need a dummyrr.
     */
    if(ctlr->dummyrr && (ctlr->width == 1 || ctlr->width == 2)){
        if(ctlr->width == 2)
            width = 1;
        else
            width = 0;
        crda = to-1-width;
        regw(ctlr, Rbcr0, (len+1+width) & 0xFF);
        regw(ctlr, Rbcr1, ((len+1+width)>>8) & 0xFF);
        regw(ctlr, Rsar0, crda & 0xFF);
        regw(ctlr, Rsar1, (crda>>8) & 0xFF);
        regw(ctlr, Cr, Page0|RdREAD|Sta);

        for(timo=0;; timo++){
            if(timo > 10000){
                print("ether8390: dummyrr timeout; assuming nodummyrr\n");
                ctlr->dummyrr = 0;
                goto top;
            }
            crda = regr(ctlr, Crda0);
            crda |= regr(ctlr, Crda1)<<8;
            if(crda == to){
                /*
                 * Start the remote DMA write and make sure
                 * the registers are correct.
                 */
                regw(ctlr, Cr, Page0|RdWRITE|Sta);

                crda = regr(ctlr, Crda0);
                crda |= regr(ctlr, Crda1)<<8;
                if(crda != to)
                    panic("crda write %lud to %lud\n", crda, to);

                break;
```

```
            }
        }
    }
    else{
        regw(ctlr, Rsar0, to & 0xFF);
        regw(ctlr, Rsar1, (to>>8) & 0xFF);
        regw(ctlr, Rbcr0, len & 0xFF);
        regw(ctlr, Rbcr1, (len>>8) & 0xFF);
        regw(ctlr, Cr, Page0|RdWRITE|Sta);
    }

    /*
     * Pump the data into the I/O port
     * then wait for the remote DMA to finish.
     */
    rdwrite(ctlr, from, len);
    for(timo = 10000; (regr(ctlr, Isr) & Rdc) == 0 && timo; timo--)
            ;

    regw(ctlr, Isr, Rdc);
    regw(ctlr, Cr, cr);

    return (void*)to;
}
```

Uses `Cr-381` 266b, `Crda0-389` 266b, `Crda1-390` 266b, `Isr-388` 266b, `Page0-423` 267a, `Rbcr0-402` 266b, `Rbcr1-403` 266b, `RdABORT-420` 267a, `RdREAD-417` 267a, `RdWRITE-418` 267a, `Rdc-432` 267b, `Rsar0-400` 266b, `Rsar1-401` 266b, `Sta-412` 267a, `Txp-413` 267a, `rdwrite()` 205a, `regr` 202b, and `regw` 202a.

⟨*function* `rdwrite` 205a⟩≡ (266a)

```
static void
rdwrite(Dp8390* ctlr, void* from, int len)
{
    switch(ctlr->width){
    default:
        panic("dp8390 rdwrite: width %d\n", ctlr->width);
        break;

    case 2:
        outss(ctlr->data, from, len/2);
        break;

    case 1:
        outsb(ctlr->data, from, len);
        break;
    }
}
```

D.4.2 Reading

Packet reception is interrupt-driven. The DP8390 fires an interrupt when a packet arrives; the `interrupt` handler reads the ring buffer header (4 bytes: status, next page, length), copies the packet data from card memory to a kernel `Block` via remote-DMA read, and calls `etheriq` to deliver it to the appropriate `Netfile`. The receive ring wraps at `pstop`: when `nxtpkt` reaches the end, it wraps back to `pstart`.

⟨*function* `interrupt` *(*`(kernel/network/386/ether8390.c)`*)* 205b⟩≡ (269)

```
static void
interrupt(Ureg*, void* arg)
{
```

```
    Ether *ether;
    Dp8390 *ctlr;
    uchar isr, r;

    ether = arg;
    ctlr = ether->ctlr;

    /*
     * While there is something of interest,
     * clear all the interrupts and process.
     */
    ilock(ctlr);
    regw(ctlr, Imr, 0x00);
    while(isr = (regr(ctlr, Isr) & (Cnt|Ovw|Txe|Rxe|Ptx|Prx))){
        if(isr & Ovw){
            overflow(ether);
            regw(ctlr, Isr, Ovw);
            ether->overflows++;
        }

        /*
         * Packets have been received.
         * Take a spin round the ring.
         */
        if(isr & (Rxe|Prx)){
            receive(ether);
            regw(ctlr, Isr, Rxe|Prx);
        }

        /*
         * A packet completed transmission, successfully or
         * not. Start transmission on the next buffered packet,
         * and wake the output routine.
         */
        if(isr & (Txe|Ptx)){
            r = regr(ctlr, Tsr);
            if((isr & Txe) && (r & (Cdh|Fu|Crs|Abt))){
                print("dp8390: Tsr %#2.2ux", r);
                ether->oerrs++;
            }

            regw(ctlr, Isr, Txe|Ptx);

            if(isr & Ptx)
                ether->outpackets++;
            ctlr->txbusy = 0;
            txstart(ether);
        }

        if(isr & Cnt){
            ether->frames += regr(ctlr, Ref0);
            ether->crcs += regr(ctlr, Ref1);
            ether->buffs += regr(ctlr, Ref2);
            regw(ctlr, Isr, Cnt);
        }
    }
    regw(ctlr, Imr, Cnt|Ovw|Txe|Rxe|Ptx|Prx);
    iunlock(ctlr);
}
```

Uses `Abt-456` 268b, `Cdh-459` 268b, `Cnt-431` 267b, `Crs-457` 268b, `Fu-458` 268b, `Imr-407` 266b, `Isr-388` 266b, `Ovw-430` 267b, `Prx-426` 267b, `Ptx-427` 267b, `Ref0-392` 266b, `Ref1-393` 266b, `Ref2-394` 266b, `Rxe-428` 267b, `Tsr-385` 266b, `Txe-429` 267b, `overflow()` 209, `receive()` 207, `regr` 202b, `regw` 202a, and `txstart()` 203.

⟨*function* `receive`*(*`(kernel/network/386/ether8390.c)`*)* 207⟩≡ (269)

```
static void
receive(Ether* ether)
{
    Dp8390 *ctlr;
    uchar curr, *p;
    Hdr hdr;
    ulong count, data, len;
    Block *bp;

    ctlr = ether->ctlr;
    for(curr = getcurr(ctlr); ctlr->nxtpkt != curr; curr = getcurr(ctlr)){
        data = ctlr->nxtpkt*Dp8390BufSz;
        if(ctlr->ram)
            memmove(&hdr, (void*)(ether->mem+data), sizeof(Hdr));
        else
            _dp8390read(ctlr, &hdr, data, sizeof(Hdr));

        /*
         * Don't believe the upper byte count, work it
         * out from the software next-page pointer and
         * the current next-page pointer.
         */
        if(hdr.next > ctlr->nxtpkt)
            len = hdr.next - ctlr->nxtpkt - 1;
        else
            len = (ctlr->pstop-ctlr->nxtpkt) + (hdr.next-ctlr->pstart) - 1;
        if(hdr.len0 > (Dp8390BufSz-sizeof(Hdr)))
            len--;

        len = ((len<<8)|hdr.len0)-4;

        /*
         * Chip is badly scrogged, reinitialise the ring.
         */
        if(hdr.next < ctlr->pstart || hdr.next >= ctlr->pstop
          || len < 60 || len > sizeof(Etherpkt)){
            print("dp8390: H%2.2ux+%2.2ux+%2.2ux+%2.2ux,%lud\n",
                hdr.status, hdr.next, hdr.len0, hdr.len1, len);
            regw(ctlr, Cr, Page0|RdABORT|Stp);
            ringinit(ctlr);
            regw(ctlr, Cr, Page0|RdABORT|Sta);

            return;
        }

        /*
         * If it's a good packet read it in to the software buffer.
         * If the packet wraps round the hardware ring, read it in
         * two pieces.
         */
        if((hdr.status & (Fo|Fae|Crce|Prxok)) == Prxok && (bp = iallocb(len))){
            p = bp->rp;
            bp->wp = p+len;
            data += sizeof(Hdr);
```

```
            if((data+len) >= ctlr->pstop*Dp8390BufSz){
                count = ctlr->pstop*Dp8390BufSz - data;
                if(ctlr->ram)
                    memmove(p, (void*)(ether->mem+data), count);
                else
                    _dp8390read(ctlr, p, data, count);
                p += count;
                data = ctlr->pstart*Dp8390BufSz;
                len -= count;
            }
            if(len){
                if(ctlr->ram)
                    memmove(p, (void*)(ether->mem+data), len);
                else
                    _dp8390read(ctlr, p, data, len);
            }

            /*
             * Copy the packet to whoever wants it.
             */
            etheriq(ether, bp, 1);
        }

        /*
         * Finished with this packet, update the
         * hardware and software ring pointers.
         */
        ctlr->nxtpkt = hdr.next;

        hdr.next--;
        if(hdr.next < ctlr->pstart)
            hdr.next = ctlr->pstop-1;
        regw(ctlr, Bnry, hdr.next);
    }
}
```

Uses `Bnry-384` 266b, `Cr-381` 266b, `Crce-468` 268d, `Dp8390BufSz` 265c, `Fae-469` 268d, `Fo-470` 268d, `Page0-423` 267a, `Prxok-467` 268d, `RdABORT-420` 267a, `Sta-412` 267a, `Stp-411` 267a, `_dp8390read()` 208b, `getcurr()` 208a, `regw` 202a, and `ringinit()` 200a.

⟨*function* `getcurr` 208a⟩≡ (269)

```
static uchar
getcurr(Dp8390* ctlr)
{
    uchar cr, curr;

    cr = regr(ctlr, Cr) & ~Txp;
    regw(ctlr, Cr, Page1|(~(Ps1|Ps0) & cr));
    curr = regr(ctlr, Curr);
    regw(ctlr, Cr, cr);

    return curr;
}
```

Uses `Cr-381` 266b, `Curr-409` 266b, `Page1-424` 267a, `Ps0-421` 267a, `Ps1-422` 267a, `Txp-413` 267a, `regr` 202b, and `regw` 202a.

⟨*function* `_dp8390read` 208b⟩≡ (269)

```
static void*
_dp8390read(Dp8390* ctlr, void* to, ulong from, ulong len)
{
    uchar cr;
    int timo;
```

```
    /*
     * Read some data at offset 'from' in the card's memory
     * using the DP8390 remote DMA facility, and place it at
     * 'to' in main memory, via the I/O data port.
     */
    cr = regr(ctlr, Cr) & ~Txp;
    regw(ctlr, Cr, Page0|RdABORT|Sta);
    regw(ctlr, Isr, Rdc);

    /*
     * Set up the remote DMA address and count.
     */
    len = ROUNDUP(len, ctlr->width);
    regw(ctlr, Rbcr0, len & 0xFF);
    regw(ctlr, Rbcr1, (len>>8) & 0xFF);
    regw(ctlr, Rsar0, from & 0xFF);
    regw(ctlr, Rsar1, (from>>8) & 0xFF);

    /*
     * Start the remote DMA read and suck the data
     * out of the I/O port.
     */
    regw(ctlr, Cr, Page0|RdREAD|Sta);
    rdread(ctlr, to, len);

    /*
     * Wait for the remote DMA to complete. The timeout
     * is necessary because this routine may be called on
     * a non-existent chip during initialisation and, due
     * to the miracles of the bus, it's possible to get this
     * far and still be talking to a slot full of nothing.
     */
    for(timo = 10000; (regr(ctlr, Isr) & Rdc) == 0 && timo; timo--)
            ;

    regw(ctlr, Isr, Rdc);
    regw(ctlr, Cr, cr);

    return to;
}
```

Uses `Cr`-381 266b, `Isr`-388 266b, `Page0`-423 267a, `Rbcr0`-402 266b, `Rbcr1`-403 266b, `RdABORT`-420 267a, `RdREAD`-417 267a, `Rdc`-432 267b, `Rsar0`-400 266b, `Rsar1`-401 266b, `Sta`-412 267a, `Txp`-413 267a, `rdread()` 265d, `regr` 202b, and `regw` 202a.

⟨*function* `overflow` 209⟩≡ (269)

```
static void
overflow(Ether *ether)
{
    Dp8390 *ctlr;
    uchar txp;
    int resend;

    ctlr = ether->ctlr;

    /*
     * The following procedure is taken from the DP8390[12D] datasheet,
     * it seems pretty adamant that this is what has to be done.
     */
    txp = regr(ctlr, Cr) & Txp;
    regw(ctlr, Cr, Page0|RdABORT|Stp);
```

```
    arch_delay(2);
    regw(ctlr, Rbcr0, 0);
    regw(ctlr, Rbcr1, 0);

    resend = 0;
    if(txp && (regr(ctlr, Isr) & (Txe|Ptx)) == 0)
        resend = 1;

    regw(ctlr, Tcr, LpbkNIC);
    regw(ctlr, Cr, Page0|RdABORT|Sta);
    receive(ether);
    regw(ctlr, Isr, Ovw);
    regw(ctlr, Tcr, LpbkNORMAL);

    if(resend)
        regw(ctlr, Cr, Page0|RdABORT|Txp|Sta);
}
```

Uses `Cr-381` 266b, `Isr-388` 266b, `LpbkNIC-449` 268a, `LpbkNORMAL-448` 268a, `Ovw-430` 267b, `Page0-423` 267a, `Ptx-427` 267b, `Rbcr0-402` 266b, `Rbcr1-403` 266b, `RdABORT-420` 267a, `Sta-412` 267a, `Stp-411` 267a, `Tcr-405` 266b, `Txe-429` 267b, `Txp-413` 267a, `receive()` 207, `regr` 202b, and `regw` 202a.

D.5 Advanced features

D.5.1 Multicast

Multicast support configures the DP8390's hash filter: the 64-bit multicast address register (`mf`) is a hash table where each bit corresponds to a bucket of multicast MAC addresses. Setting a bit causes the card to accept frames whose destination MAC hashes to that bucket.

⟨`Dp8390` *multicast fields* 210a⟩≡ (196)

```
uchar mar[8];    /* shadow multicast address registers */
int mref[64];   /* reference counts for multicast groups */
```

⟨*global* `reverse` 210b⟩≡ (269)

```
static uchar reverse[64];
```

⟨*function* `multicast` 210c⟩≡ (269)

```
static void
multicast(void* arg, uchar *addr, int on)
{
    Ether *ether;
    Dp8390 *ctlr;
    int i;
    ulong h;

    ether = arg;
    ctlr = ether->ctlr;
    if(reverse[1] == 0){
        for(i = 0; i < 64; i++)
            reverse[i] = ((i&1)<<5) | ((i&2)<<3) | ((i&4)<<1)
                   | ((i&8)>>1) | ((i&16)>>3) | ((i&32)>>5);
    }

    /*
     *  change filter bits
     */
    h = ethercrc(addr, 6);
    ilock(ctlr);
```

```
      setbit(ctlr, reverse[h&0x3f], on);
      setfilter(ether, ctlr);
      iunlock(ctlr);
  }
```
Uses `reverse-476` 210b, `setbit()` 211b, and `setfilter()` 211c.

⟨*global* `allmar` 211a⟩≡ (269)
```
  static uchar allmar[8] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
```

⟨*function* `setbit` 211b⟩≡ (269)
```
  static void
  setbit(Dp8390 *ctlr, int bit, int on)
  {
      int i, h;

      i = bit/8;
      h = bit%8;
      if(on){
          if(++(ctlr->mref[bit]) == 1)
              ctlr->mar[i] |= 1<<h;
      } else {
          if(--(ctlr->mref[bit]) <= 0){
              ctlr->mref[bit] = 0;
              ctlr->mar[i] &= ~(1<<h);
          }
      }
  }
```

⟨*function* `setfilter` 211c⟩≡ (269)
```
  static void
  setfilter(Ether *ether, Dp8390 *ctlr)
  {
      uchar r, cr;
      int i;
      uchar *mar;

      r = Ab;
      mar = 0;
      if(ether->prom){
          r |= Pro|Am;
          mar = allmar;
      } else if(ether->nmaddr){
          r |= Am;
          mar = ctlr->mar;
      }
      if(mar){
          cr = regr(ctlr, Cr) & ~Txp;
          regw(ctlr, Cr, Page1|(~(Ps1|Ps0) & cr));
          for(i = 0; i < 8; i++)
              regw(ctlr, Mar0+i, *(mar++));
          regw(ctlr, Cr, cr);
      }
      regw(ctlr, Rcr, r);
  }
```
Uses `Ab-463` 268c, `Am-464` 268c, `Cr-381` 266b, `Mar0-410` 266b, `Page1-424` 267a, `Pro-465` 268c, `Ps0-421` 267a, `Ps1-422` 267a, `Rcr-404` 266b, `Txp-413` 267a, `allmar-475` 211a, `regr` 202b, and `regw` 202a.

D.5.2 Promiscuous mode

⟨*function* promiscuous 212a⟩≡ (269)
```
static void
promiscuous(void *arg, int )
{
    Ether *ether;
    Dp8390 *ctlr;

    ether = arg;
    ctlr = ether->ctlr;

    ilock(ctlr);
    setfilter(ether, ctlr);
    iunlock(ctlr);
}
```
Uses `setfilter()` 211c.

D.5.3 Power managment

D.5.4 Plug and play

PCI/ISA plug-and-play detection scans the PCI bus for known NE2000-compatible device/vendor IDs. `ne2000pnp` iterates over PCI devices matching the NE2000 class code and configures the I/O port and IRQ from the PCI configuration space, avoiding the need for manual hardware configuration.

⟨*struct* Ctlr*(*(kernel/network/386/ether2000.c)*)* 212b⟩≡ (270b)
```
struct Ctlr {
    Pcidev* pcidev;
    Ctlr* next;
    int active;
};
```

⟨*global* ctlrhead*(*(kernel/network/386/ether2000.c)*)* 212c⟩≡ (270b)
```
static Ctlr* ctlrhead;
```

⟨*global* ctlrtail*(*(kernel/network/386/ether2000.c)*)* 212d⟩≡ (270b)
```
static Ctlr* ctlrtail;
```

⟨*function* ne2000pnp 212e⟩≡ (270b)
```
static void
ne2000pnp(Ether* edev)
{
    int i, id;
    Pcidev *p;
    Ctlr *ctlr;

    /*
     * Make a list of all ethernet controllers
     * if not already done.
     */
    if(ctlrhead == nil){
        p = nil;
        while(p = pcimatch(p, 0, 0)){
            if(p->ccrb != 0x02 || p->ccru != 0)
                continue;
            ctlr = malloc(sizeof(Ctlr));
            if(ctlr == nil)
                error(Enomem);
```

```
            ctlr->pcidev = p;

            if(ctlrhead != nil)
                ctlrtail->next = ctlr;
            else
                ctlrhead = ctlr;
            ctlrtail = ctlr;
        }
    }

    /*
     * Is it a card with an unrecognised vid+did?
     * Normally a search is made through all the found controllers
     * for one which matches any of the known vid+did pairs.
     * If a vid+did pair is specified a search is made for that
     * specific controller only.
     */
    id = 0;
    for(i = 0; i < edev->nopt; i++){
        if(cistrncmp(edev->opt[i], "id=", 3) == 0)
            id = strtol(&edev->opt[i][3], nil, 0);
    }

    if(id != 0)
        ne2000match(edev, id);
    else for(i = 0; ne2000pci[i].name; i++){
        if(ne2000match(edev, ne2000pci[i].id) != nil)
            break;
    }
}
```

Uses `ctlrhead-378` 212c, `ctlrtail-379` 212d, `ne2000match()` 213b, and `ne2000pci-380` 213a.

⟨*global* `ne2000pci` 213a⟩≡ (270b)

```
static struct {
    char* name;
    int id;
} ne2000pci[] = {
    { "Realtek 8029", (0x8029<<16)|0x10EC, },
    { "Winbond 89C940", (0x0940<<16)|0x1050, },
    { nil },
};
```

Uses `__anon_struct_46` 213a.

⟨*function* `ne2000match` 213b⟩≡ (270b)

```
static Ctlr*
ne2000match(Ether* edev, int id)
{
    int port;
    Pcidev *p;
    Ctlr *ctlr;

    /*
     * Any adapter matches if no edev->port is supplied,
     * otherwise the ports must match.
     */
    for(ctlr = ctlrhead; ctlr != nil; ctlr = ctlr->next){
        if(ctlr->active)
            continue;
        p = ctlr->pcidev;
        if(((p->did<<16)|p->vid) != id)
```

```
            continue;
        port = p->mem[0].bar & ~0x01;
        if(edev->port != 0 && edev->port != port)
            continue;

        /*
         * It suffices to fill these in,
         * the rest is gleaned from the card.
         */
        edev->port = port;
        edev->irq = p->intl;

        ctlr->active = 1;

        return ctlr;
    }

    return nil;
}
```

Uses `ctlrhead-378` 212c.

Appendix E

Utilities

E.1 Globbing

Appendix F

Extra Mediums

F.1 Null medium

The null medium discards all packets—its `bwrite` frees the block immediately. It is used for interfaces that need to exist in the routing table but don't actually transmit. `nullbind` simply records the device name without opening any hardware.

⟨*global* `nullmedium` 216a⟩≡ (355b)

```
Medium nullmedium =
{
    .name=      "null",

    .bind=      nullbind,
    .unbind=    nullunbind,

    .bwrite=    nullbwrite,
};
```

Uses `nullbind()` 216b, `nullbwrite()` 216d, and `nullunbind()` 216c.

⟨*function* `nullbind` 216b⟩≡ (355b)

```
static void
nullbind(Ipifc*, int, char**)
{
    error("cannot bind null device");
}
```

⟨*function* `nullunbind` 216c⟩≡ (355b)

```
static void
nullunbind(Ipifc*)
{
}
```

⟨*function* `nullbwrite` 216d⟩≡ (355b)

```
static void
nullbwrite(Ipifc*, Block*, int, uchar*)
{
    error("nullbwrite");
}
```

⟨*function* `nullmediumlink` 216e⟩≡ (355b)

```
void
nullmediumlink(void)
{
    addipmedium(&nullmedium);
}
```

Uses `addipmedium()` 27c and `nullmedium` 216a.

F.2 Loopback medium

The loopback medium implements local-to-local communication: packets written via `loopbackbwrite` are immediately fed back to `ipiput4` on the same interface, without ever touching hardware. This is how a machine communicates with itself (address 127.0.0.1). The loopback medium has no MAC address and no header overhead.

⟨*global* `loopbackmedium` 217a⟩≡ (356a)
```
Medium loopbackmedium =
{
    .name=      "loopback",

    .hsize=     0,
    .mintu=     0,
    .maxtu=     Maxtu,
    .maclen=    0,

    .bind=      loopbackbind,
    .unbind=    loopbackunbind,
    .bwrite=    loopbackbwrite,
};
```
Uses `Maxtu-263` 217b, `loopbackbind()` 217e, `loopbackbwrite()` 218c, and `loopbackunbind()` 218a.

⟨*enum* `_anon_ (kernel/network/ip/loopbackmedium.c)` 217b⟩≡ (356a)
```
enum
{
    Maxtu=  16*1024,
};
```

⟨*function* `loopbackmediumlink` 217c⟩≡ (356a)
```
void
loopbackmediumlink(void)
{
    addipmedium(&loopbackmedium);
}
```
Uses `addipmedium()` 27c and `loopbackmedium` 217a.

⟨*struct* `LB` 217d⟩≡ (356a)
```
struct LB
{
    Queue   *q;
    Fs  *f;
    Proc    *readp;
};
```

⟨*function* `loopbackbind` 217e⟩≡ (356a)
```
static void
loopbackbind(Ipifc *ifc, int, char**)
{
    LB *lb;

    lb = smalloc(sizeof(LB));
    lb->f = ifc->conv->p->f;
    lb->q = qopen(1024*1024, Qmsg, nil, nil);
    ifc->arg = lb;
    ifc->mbps = 1000;

    kproc("loopbackread", loopbackread, ifc);

}
```
Uses `loopbackread()` 218b.

⟨*function* loopbackunbind 218a⟩≡ (356a)

```
static void
loopbackunbind(Ipifc *ifc)
{
    LB *lb = ifc->arg;

    if(lb->readp)
        postnote(lb->readp, 1, "unbind", 0);

    /* wait for reader to die */
    while(lb->readp != nil)
        tsleep(&up->sleepr, returnfalse, 0, 300);

    /* clean up */
    qfree(lb->q);
    free(lb);
}
```

⟨*function* loopbackread 218b⟩≡ (356a)

```
static void
loopbackread(void *a)
{
    Ipifc *ifc;
    Block *bp;
    LB *lb;

    ifc = a;
    lb = ifc->arg;
    lb->readp = up; /* hide identity under a rock for unbind */
    if(waserror()){
        lb->readp = 0;
        pexit("hangup", 1);
    }
    for(;;){
        bp = qbread(lb->q, Maxtu);
        if(bp == nil)
            continue;
        ifc->in++;
        if(!canrlock(ifc)){
            freeb(bp);
            continue;
        }
        if(waserror()){
            runlock(ifc);
            nexterror();
        }
        if(ifc->lifc == nil)
            freeb(bp);
        else
            ipiput4(lb->f, ifc, bp);
        runlock(ifc);
        poperror();
    }
}
```

Uses `Maxtu-263` 217b and `ipiput4()` 98.

⟨*function* loopbackbwrite 218c⟩≡ (356a)

```
static void
loopbackbwrite(Ipifc *ifc, Block *bp, int, uchar*)
{
```

```
    LB *lb;

    lb = ifc->arg;
    if(qpass(lb->q, bp) < 0)
        ifc->outerr++;
    ifc->out++;
}
```

F.3 Point to point serial line

F.4 Token ring

Appendix G

Extra Protocols

G.1 ICMP

⟨Proto(kernel) *conversation methods* 220⟩+≡ (32f) ◁54a

```
  void    (*advise)(Proto*, Block*, char*);
```

G.2 RUDP

G.3 GRE

G.4 ESP

G.5 Datakit and URP

Appendix H

Extra Applications

H.1 Remote login, rlogin

H.2 Serving files, FTP

H.3 Serving documents, HTTP

H.4 Serving mails, SMTP

H.5 Serving news, NNTP

Appendix I

Extra Code

I.1 include/net/

I.1.1 include/net/ip.h

⟨*enum* _anon_ 222a⟩≡ (222c)

```
enum
{
    ⟨constant IPaddrlen 20c⟩
    ⟨constant IPv4addrlen 20a⟩
    ⟨constant IPv4off 21a⟩

    IPllen=  4,
    IPV4HDR_LEN= 20,

    /* vihl & vcf[0] values */
    IP_VER4=  0x40,
    IP_VER6= 0x60,
};
```

⟨*constant* Udphdrsize 222b⟩≡ (222c)

```
/*
 *  user level udp headers with control message "headers"
 */
#define Udphdrsize 52 /* size of a Udphdr */
```

⟨include/net/ip.h 222c⟩≡

```
#pragma src "/sys/src/libip"
#pragma lib "libip.a"

⟨enum _anon_ 222a⟩

⟨typedef ipv4 20b⟩
⟨typedef ipaddr 20d⟩
⟨typedef iplong 23a⟩


// forward decl
typedef struct Ipifc Ipifc;
typedef struct Iplifc Iplifc;
typedef struct Ipv6rp Ipv6rp;
typedef struct Ip6hdr Ip6hdr;
typedef struct Icmp6hdr Icmp6hdr;
typedef struct Udphdr Udphdr;
```

```
/*
 *  for reading /net/ipifc
 */

⟨struct Iplifc (user) 24e⟩

⟨struct Ipv6rp 502c⟩

⟨struct Ipifc (user) 24a⟩

⟨macro ISIPV6MCAST 502d⟩
⟨macro ISIPV6LINKLOCAL 502e⟩

⟨enum _anon_ (include/net/ip.h) 502f⟩


⟨struct Ip6hdr 503a⟩

⟨struct Icmp6hdr 503b⟩

⟨constant Udphdrsize 222b⟩

⟨struct Udphdr (user) 138f⟩

uchar* defmask(ipaddr);
void maskip(ipaddr, ipaddr, ipaddr);
int  eipfmt(Fmt*);
bool isv4(ipaddr);
vlong parseip(uchar*, char*);
vlong parseipmask(uchar*, char*);
char* v4parseip(uchar*, char*);
//char* v4parsecidr(uchar*, uchar*, char*);
int parseether(uchar*, char*);
int myipaddr(uchar*, char*);
int myetheraddr(uchar*, char*);
int equivip4(uchar*, uchar*);
int equivip6(uchar*, uchar*);

Ipifc* readipifc(char*, Ipifc*, int);

void hnputs(void*, ushort);
void hnputl(void*, uint);
void hnputv(void*, uvlong);
ushort nhgets(void*);
uint nhgetl(void*);
uvlong nhgetv(void*);

ushort ptclbsum(uchar*, int);

int  v6tov4(uchar*, uchar*);
void v4tov6(uchar*, uchar*);

⟨macro ipcmp 90a⟩
⟨macro ipmove 90b⟩

extern ipaddr IPv4bcast;
extern ipaddr IPv4bcastobs;
extern ipaddr IPv4allsys;
extern ipaddr IPv4allrouter;
extern ipaddr IPnoaddr;
```

```
extern ipaddr v4prefix;
extern ipaddr IPallbits;

⟨macro CLASS 22a⟩

#pragma varargck type "I" uchar*
#pragma varargck type "V" uchar*
#pragma varargck type "E" uchar*
#pragma varargck type "M" uchar*
```

I.2 libip/

I.2.1 libip/parseether.c

⟨libip/parseether.c 224a⟩≡

```
#include <u.h>
#include <libc.h>

⟨function parseether 85a⟩
```

I.2.2 libip/parseip.c

⟨*function* ipcharok 224b⟩≡ (225a)

```
static int
ipcharok(int c)
{
    return c == '.' || c == ':' || isascii(c) && isxdigit(c);
}
```

⟨*function* delimchar 224c⟩≡ (225a)

```
static bool
delimchar(int c)
{
    if(c == '\0')
        return true;
    if(c == '.' || c == ':' || isascii(c) && isalnum(c))
        return false;
    return true;
}
```

⟨*function* parseipmask 224d⟩≡ (225a)

```
/*
 *  hack to allow ip v4 masks to be entered in the old
 *  style
 */
vlong
parseipmask(uchar *to, char *from)
{
    int i, w;
    vlong x;
    uchar *p;

    if(*from == '/'){
        /* as a number of prefix bits */
        i = atoi(from+1);
        if(i < 0)
            i = 0;
```

```
        if(i > 128)
            i = 128;
        w = i;
        memset(to, 0, IPaddrlen);
        for(p = to; i >= 8; i -= 8)
            *p++ = 0xff;
        if(i > 0)
            *p = ~((1<<(8-i))-1);
        x = nhgetl(to+IPv4off);
        /*
         * identify as ipv6 if the mask is inexpressible as a v4 mask
         * (because it has too few mask bits).  Arguably, we could
         * always return 6 here.
         */
        if (w < 8*(IPaddrlen-IPv4addrlen))
            return 6;
    } else {
        /* as a straight v4 bit mask */
        x = parseip(to, from);
        if (x != -1)
            x = (ulong)nhgetl(to + IPv4off);
        if(memcmp(to, v4prefix, IPv4off) == 0)
            memset(to, 0xff, IPv4off);
    }
    return x;
}
```

Uses `IPaddrlen` 20c, `IPv4addrlen` 20a, `IPv4off` 21a, `parseip()` 89, and `v4prefix` 20e.

⟨`libip/parseip.c` 225a⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
#include <ip.h>
```

⟨*function* `v4parseip` 88⟩

⟨*function* `ipcharok` 224b⟩

⟨*function* `delimchar` 224c⟩

⟨*function* `parseip` 89⟩

⟨*function* `parseipmask` 224d⟩

I.2.3 `libip/myetheraddr.c`

⟨`libip/myetheraddr.c` 225b⟩≡

```
#include <u.h>
#include <libc.h>
#include <ip.h>
```

⟨*function* `myetheraddr` 85b⟩

I.2.4 `libip/myipaddr.c`

⟨*global* `loopbacknet` 225c⟩≡ (226b)

```
static ipaddr loopbacknet = {
```

```
      0, 0, 0, 0,
      0, 0, 0, 0,
      0, 0, 0xff, 0xff,
      127, 0, 0, 0
  };
```

⟨*global* loopbackmask 226a⟩≡ (226b)

```
  static ipaddr loopbackmask = {
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff,
      0xff, 0, 0, 0
  };
```

⟨libip/myipaddr.c 226b⟩≡

```
  #include <u.h>
  #include <libc.h>
  #include <ip.h>

  ⟨global loopbacknet 225c⟩
  ⟨global loopbackmask 226a⟩

  ⟨function myipaddr 76c⟩
```

I.2.5 libip/eipfmt.c

⟨*enum* _anon_ (libip/eipfmt.c) 226c⟩≡ (226e)

```
  enum
  {
      Isprefix= 16,
  };
```

⟨*global* prefixvals 226d⟩≡ (226e)

```
  uchar prefixvals[256] =
  {
  [0x00] 0 | Isprefix,
  [0x80] 1 | Isprefix,
  [0xC0] 2 | Isprefix,
  [0xE0] 3 | Isprefix,
  [0xF0] 4 | Isprefix,
  [0xF8] 5 | Isprefix,
  [0xFC] 6 | Isprefix,
  [0xFE] 7 | Isprefix,
  [0xFF] 8 | Isprefix,
  };
```

Uses Isprefix-3 226c.

⟨libip/eipfmt.c 226e⟩≡

```
  #include <u.h>
  #include <libc.h>
  #include <ip.h>

  ⟨enum _anon_ (libip/eipfmt.c) 226c⟩

  ⟨global prefixvals 226d⟩

  ⟨function eipfmt 186⟩
```

I.2.6 `libip/equivip.c`

```
⟨libip/equivip.c 227a⟩≡
  #include <u.h>
  #include <libc.h>
  #include <ip.h>

  ⟨function equivip4 90c⟩

  ⟨function equivip6 504a⟩
```

I.2.7 `libip/ipaux.c`

```
⟨global IPv4allsys 227b⟩≡                                                    (227e)
  ipaddr IPv4allsys = {
      0, 0, 0, 0,
      0, 0, 0, 0,
      0, 0, 0xff, 0xff,

      0xe0, 0, 0, 0x01
  };

⟨global IPv4allrouter 227c⟩≡                                                 (227e)
  ipaddr IPv4allrouter = {
      0, 0, 0, 0,
      0, 0, 0, 0,
      0, 0, 0xff, 0xff,

      0xe0, 0, 0, 0x02
  };

⟨global IPallbits 227d⟩≡                                                     (227e)
  ipaddr IPallbits = {
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff
  };

⟨libip/ipaux.c 227e⟩≡
  #include <u.h>
  #include <libc.h>
  #include <ip.h>

  ⟨global IPv4bcast 179b⟩
  ⟨global IPv4allsys 227b⟩
  ⟨global IPv4allrouter 227c⟩
  ⟨global IPallbits 227d⟩
  ⟨global IPnoaddr 23b⟩

  ⟨global v4prefix 20e⟩

  ⟨function isv4 21b⟩

  ⟨function v4tov6 21c⟩

  ⟨function v6tov4 21d⟩
```

I.2.8 libip/bo.c

⟨*function* hnputv 228a⟩≡ (229c)

```
void
hnputv(void *p, uvlong v)
{
	uchar *a;

	a = p;
	a[0] = v>>56;
	a[1] = v>>48;
	a[2] = v>>40;
	a[3] = v>>32;
	a[4] = v>>24;
	a[5] = v>>16;
	a[6] = v>>8;
	a[7] = v;
}
```

⟨*function* hnputl 228b⟩≡ (229c)

```
void
hnputl(void *p, uint v)
{
	uchar *a;

	a = p;
	a[0] = v>>24;
	a[1] = v>>16;
	a[2] = v>>8;
	a[3] = v;
}
```

⟨*function* hnputs 228c⟩≡ (229c)

```
void
hnputs(void *p, ushort v)
{
	uchar *a;

	a = p;
	a[0] = v>>8;
	a[1] = v;
}
```

⟨*function* nhgetv 228d⟩≡ (229c)

```
uvlong
nhgetv(void *p)
{
	uchar *a;
	uvlong v;

	a = p;
	v = (uvlong)a[0]<<56;
	v |= (uvlong)a[1]<<48;
	v |= (uvlong)a[2]<<40;
	v |= (uvlong)a[3]<<32;
	v |= a[4]<<24;
	v |= a[5]<<16;
	v |= a[6]<<8;
	v |= a[7]<<0;
	return v;
}
```

⟨*function* nhgetl 229a⟩≡ (229c)

```
uint
nhgetl(void *p)
{
    uchar *a;

    a = p;
    return (a[0]<<24)|(a[1]<<16)|(a[2]<<8)|(a[3]<<0);
}
```

⟨*function* nhgets 229b⟩≡ (229c)

```
ushort
nhgets(void *p)
{
    uchar *a;

    a = p;
    return (a[0]<<8)|(a[1]<<0);
}
```

⟨libip/bo.c 229c⟩≡

```
#include <u.h>
#include <libc.h>
#include <ip.h>

⟨function hnputv 228a⟩

⟨function hnputl 228b⟩

⟨function hnputs 228c⟩

⟨function nhgetv 228d⟩

⟨function nhgetl 229a⟩

⟨function nhgets 229b⟩
```

I.2.9 libip/classmask.c

⟨libip/classmask.c 229d⟩≡

```
#include <u.h>
#include <libc.h>
#include <ip.h>

⟨global classmask 22c⟩

⟨global v6loopback 504b⟩

⟨global v6linklocal 504c⟩
⟨global v6linklocalmask 504d⟩
⟨global v6llpreflen 504e⟩

⟨global v6multicast 504f⟩
⟨global v6multicastmask 504g⟩
⟨global v6mcpreflen 505a⟩

⟨global v6solicitednode 505b⟩
⟨global v6solicitednodemask 505c⟩
⟨global v6snpreflen 505d⟩
```

⟨*function* `defmask` 22b⟩

⟨*function* `maskip` 22d⟩

I.2.10 libip/ptclbsum.c

⟨*global* `endian` 230a⟩≡ (231a)

```
static short endian = 1;
```

Uses `endian-16` 230a.

⟨*global* `aendian` 230b⟩≡ (231a)

```
static uchar* aendian = (uchar*)&endian;
```

Uses `aendian-17` 230b and `endian-16` 230a.

⟨*constant* `LITTLE` 230c⟩≡ (231a)

```
#define LITTLE *aendian
```

⟨*function* `ptclbsum` 230d⟩≡ (231a)

```
ushort
ptclbsum(uchar *addr, int len)
{
    ulong losum, hisum, mdsum, x;
    ulong t1, t2;

    losum = 0;
    hisum = 0;
    mdsum = 0;

    x = 0;
    if((uintptr)addr & 1) {
        if(len) {
            hisum += addr[0];
            len--;
            addr++;
        }
        x = 1;
    }
    while(len >= 16) {
        t1 = *(ushort*)(addr+0);
        t2 = *(ushort*)(addr+2); mdsum += t1;
        t1 = *(ushort*)(addr+4); mdsum += t2;
        t2 = *(ushort*)(addr+6); mdsum += t1;
        t1 = *(ushort*)(addr+8); mdsum += t2;
        t2 = *(ushort*)(addr+10); mdsum += t1;
        t1 = *(ushort*)(addr+12); mdsum += t2;
        t2 = *(ushort*)(addr+14); mdsum += t1;
        mdsum += t2;
        len -= 16;
        addr += 16;
    }
    while(len >= 2) {
        mdsum += *(ushort*)addr;
        len -= 2;
        addr += 2;
    }
    if(x) {
        if(len)
            losum += addr[0];
```

```
        if(LITTLE)
            losum += mdsum;
        else
            hisum += mdsum;
    } else {
        if(len)
            hisum += addr[0];
        if(LITTLE)
            hisum += mdsum;
        else
            losum += mdsum;
    }

    losum += hisum >> 8;
    losum += (hisum & 0xff) << 8;
    while(hisum = losum>>16)
        losum = hisum + (losum & 0xffff);

    return losum & 0xffff;
}
```

Uses LITTLE-18 230c.

⟨libip/ptclbsum.c 231a⟩≡

```
#include <u.h>
#include <libc.h>
#include <ip.h>

⟨global endian 230a⟩
⟨global aendian 230b⟩
⟨constant LITTLE 230c⟩

⟨function ptclbsum 230d⟩
```

I.2.11 libip/readipifc.c

⟨libip/readipifc.c 231b⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
#include <ip.h>

⟨function findfield 76b⟩

⟨function _readipifc 74⟩

⟨function _freeifc 76a⟩

⟨function readipifc 73c⟩
```

I.3 libip/tests/

I.3.1 libip/tests/testreadipifc.c

⟨libip/tests/testreadipifc.c 231c⟩≡

```
#include <u.h>
#include <libc.h>
#include <ip.h>
```

⟨*function* main 188a⟩

I.4 kernel/network/ip/

I.4.1 kernel/network/ip/ip.h

⟨*constant* Nhash 232a⟩≡ (232b)

```
Nhash=    64,
```

⟨*enum* _anon_ (kernel/network/ip/ip.h) 232b⟩≡ (234b)

```
enum
{
  Addrlen=  64,
  ⟨constant Maxproto 29e⟩
  ⟨constant Nhash 232a⟩
  Maxincall=  32, /* max. conn.s in listen q not accepted yet */
  ⟨constant Nchans 135b⟩
  MAClen=   16,   /* longest mac address */

  ⟨constant MAXTTL 81g⟩
  DFLTTOS=  0,

  ⟨constant IPaddrlen 20c⟩
  ⟨constant IPv4addrlen 20a⟩
  ⟨constant IPv4off 21a⟩
  IPllen=   4,

  ⟨constant IP_VER4 94a⟩
  ⟨constant IP_HLEN4 94b⟩
  IP_VER6=  0x60,

  IP_DF=    0x4000,   /* v4: Don't fragment */
  IP_MF=    0x2000,   /* v4: More fragments */
  IP4HDR=   20,   /* sizeof(Ip4hdr) */

  ⟨constant IP_MAX 97a⟩

  ⟨constant Lroot 39b⟩

  Maxpath = 64,
};
```

⟨*struct* Ipfrag 232c⟩≡ (234b)

```
//@Scheck: used only for its macro below, could maybe simplify?
struct Ipfrag
{
  ushort  foff;
  ushort  flen;

  uchar payload[];
};
```

⟨*constant* IPFRAGSZ 232d⟩≡ (234b)

```
#define IPFRAGSZ offsetof(Ipfrag, payload[0])
```

⟨*struct* Routerparams 233a⟩≡ (234b)
```
/* default values, one per stack */
struct Routerparams {
  int mflag;    /* flag: managed address configuration */
  int oflag;    /* flag: other stateful configuration */
  int   maxraint; /* max. router adv interval (ms) */
  int minraint; /* min. router adv interval (ms) */
  int linkmtu;  /* mtu options */
  int reachtime;  /* reachable time */
  int rxmitra;  /* retransmit interval */
  int ttl;    /* cur hop count limit */
  int routerlt; /* router lifetime */
};
```

⟨*struct* Hostparams 233b⟩≡ (234b)
```
struct Hostparams {
  int rxmithost;
};
```

⟨*enum* _anon_ (kernel/network/ip/ip.h)4 233c⟩≡ (234b)
```
enum
{
  ⟨constant Nipht 135e⟩
};
```

⟨*enum* _anon_ (kernel/network/ip/ip.h)5 233d⟩≡ (234b)
```
/*
 *  logging
 */
enum
{
  Logip=    1<<1,
  Logtcp=   1<<2,
  Logfs=    1<<3,
  Logicmp=  1<<5,
  Logudp=   1<<6,
  Logcompress=  1<<7,
  Loggre=   1<<9,
  Logppp=   1<<10,
  Logtcprxmt= 1<<11,
  Logigmp=  1<<12,
  Logudpmsg=  1<<13,
  Logipmsg= 1<<14,
  Logrudp=  1<<15,
  Logrudpmsg= 1<<16,
  Logesp=   1<<17,
  Logtcpwin=  1<<18,
};
```

⟨*struct* Routewalk 233e⟩≡ (234b)
```
struct Routewalk
{
  int o;
  int h;
  char* p;
  char* e;
  void* state;
  void  (*walk)(Route*, Routewalk*);
};
```

⟨*constant* NOW 234a⟩≡ (234b)

```
#define NOW TK2MS(CPUS(0)->ticks)
```

⟨kernel/network/ip/ip.h 234b⟩≡

```
// coupling: include/ip.h
// This file references also code in lib_networking (linked with the kernel).
// Those functions are also exported in include/ip.h.
// Some types are duplicated with include/ip.h.

// forward decls
typedef struct  Conv  Conv;
typedef struct  Fragment4 Fragment4;
typedef struct  Fragment6 Fragment6;
typedef struct  Fs  Fs;
typedef struct  IP  IP;
typedef struct  IPaux IPaux;
typedef struct  Ip4hdr  Ip4hdr;
typedef struct  Ipfrag  Ipfrag;
typedef struct  Ipself  Ipself;
typedef struct  Ipselftab Ipselftab;
typedef struct  Iplink  Iplink;
typedef struct  Iplifc  Iplifc;
typedef struct  Ipmulti Ipmulti;
typedef struct  Ipifc Ipifc;
typedef struct  Iphash  Iphash;
typedef struct  Ipht  Ipht;
typedef struct  Netlog  Netlog;
typedef struct  Medium  Medium;
typedef struct  Proto Proto;
typedef struct  Arpent  Arpent;
typedef struct  Arp Arp;
typedef struct  Route Route;
typedef struct  Routerparams  Routerparams;
typedef struct  Hostparams  Hostparams;
typedef struct  v6router  v6router;
typedef struct  v6params  v6params;

#pragma incomplete Arp
#pragma incomplete Ipself
#pragma incomplete Ipselftab
#pragma incomplete IP
#pragma incomplete Netlog
```

⟨*enum* _anon_ (kernel/network/ip/ip.h) 232b⟩

⟨*enum* ip_version 21g⟩

⟨*typedef ipv4* 20b⟩
⟨*typedef ipaddr* 20d⟩
⟨*typedef iplong* 23a⟩
⟨*typedef ipv4or6* 21e⟩
⟨*typedef ipv4p* 21f⟩

⟨*enum* conversation_state 34b⟩

⟨*enum* _anon_ (kernel/network/ip/ip.h)3 31a⟩

⟨*struct* Fragment4 30d⟩

```
⟨struct Fragment6 505h⟩

⟨struct Ipfrag 232c⟩

⟨constant IPFRAGSZ 232d⟩

⟨struct IP (kernel) 30c⟩

⟨struct Ip4hdr 23e⟩

⟨struct Conv (kernel) 33e⟩

⟨struct Medium (kernel) 26b⟩

⟨struct Iplifc (kernel) 25e⟩

⟨struct Iplink 127a⟩

/* rfc 2461, pp.40|43. */

⟨struct Routerparams 233a⟩

⟨struct Hostparams 233b⟩

⟨struct Ipifc (kernel) 25b⟩

⟨struct Ipmulti 179c⟩

⟨enum _anon_ (kernel/network/ip/ip.h)4 233c⟩

⟨enum matchtype 136b⟩

⟨struct Iphash 136a⟩
⟨struct Ipht 135d⟩

void iphtadd(Ipht*, Conv*);
void iphtrem(Ipht*, Conv*);
Conv* iphtlook(Ipht *ht, uchar *sa, ushort sp, uchar *da, ushort dp);

⟨struct Proto (kernel) 32a⟩


⟨struct Fs (kernel) 29d⟩

⟨struct v6router 506a⟩

⟨struct v6params 506b⟩

int Fsconnected(Conv*, char*);
Conv* Fsnewcall(Conv*, uchar*, ushort, uchar*, ushort, uchar);
//int Fspcolstats(char*, int);
int Fsproto(Fs*, Proto*);
//int Fsbuiltinproto(Fs*, uchar);
//Conv* Fsprotoclone(Proto*, char*);
Proto*  Fsrcvpcol(Fs*, uchar);
Proto*  Fsrcvpcolx(Fs*, uchar);
char* Fsstdconnect(Conv*, char**, int);
char* Fsstdannounce(Conv*, char**, int);
//char* Fsstdbind(Conv*, char**, int);
ulong scalednconv(void);
```

```
//void  closeconv(Conv*);
⟨enum _anon_ (kernel/network/ip/ip.h)5 233d⟩

void  netloginit(Fs*);
void  netlogopen(Fs*);
void  netlogclose(Fs*);
void  netlogctl(Fs*, char*, int);
long  netlogread(Fs*, void*, ulong, long);
void  netlog(Fs*, int, char*, ...);
//void  ifcloginit(Fs*);
//long  ifclogread(Fs*, Chan *,void*, ulong, long);
//void  ifclog(Fs*, uchar *, int);
//void  ifclogopen(Fs*, Chan*);
//void  ifclogclose(Fs*, Chan*);

#pragma varargck argpos netlog  3

/*
 *  iproute.c
 */
typedef struct RouteTree RouteTree;
typedef struct Routewalk Routewalk;
typedef struct V4route V4route;
typedef struct V6route V6route;

⟨enum _anon_ (kernel/network/ip/ip.h)6 38e⟩

⟨struct Routewalk 233e⟩

⟨struct RouteTree (kernel) 38b⟩

⟨struct V4route 38a⟩

⟨struct V6route 506c⟩

⟨struct Route (kernel) 37e⟩
extern void v4addroute(Fs *f, char *tag, uchar *a, uchar *mask, uchar *gate, int type);
extern void v4delroute(Fs *f, uchar *a, uchar *mask, int dolock);
extern Route* v4lookup(Fs *f, uchar *a, Conv *c);

extern void v6addroute(Fs *f, char *tag, uchar *a, uchar *mask, uchar *gate, int type);
extern void v6delroute(Fs *f, uchar *a, uchar *mask, int dolock);
extern Route* v6lookup(Fs *f, uchar *a, Conv *c);

extern long routeread(Fs *f, char*, ulong, int);
extern long routewrite(Fs *f, Chan*, char*, int);
extern void routetype(int, char*);
//extern void ipwalkroutes(Fs*, Routewalk*);
//extern void convroute(Route*, uchar*, uchar*, uchar*, char*, int*);

/*
 *  devip.c
 */

⟨struct IPaux 41a⟩

extern IPaux* newipaux(char*, char*);

/*
 *  arp.c
```

```
 */
⟨struct Arpent 106d⟩

extern void arpinit(Fs*);
extern int  arpread(Arp*, char*, ulong, int);
extern int  arpwrite(Fs*, char*, int);
extern Arpent*  arpget(Arp*, Block *bp, int version, Ipifc *ifc, uchar *ip, uchar *h);
extern void arprelease(Arp*, Arpent *a);
extern Block* arpresolve(Arp*, Arpent *a, Medium *type, uchar *mac);
extern void arpenter(Fs*, int version, uchar *ip, uchar *mac, int len, int norefresh);

/*
 * ipaux.c
 */

//extern int  myetheraddr(uchar*, char*);
extern vlong  parseip(uchar*, char*);
extern vlong  parseipmask(uchar*, char*);
//extern char*  v4parseip(uchar*, char*);
extern void maskip(uchar *from, uchar *mask, uchar *to);
extern int  parsemac(uchar *to, char *from, int len);
extern uchar* defmask(uchar*);
extern int  isv4(uchar*);
extern void v4tov6(uchar *v6, uchar *v4);
extern int  v6tov4(uchar *v4, uchar *v6);
extern int  eipfmt(Fmt*);

⟨macro ipmove((kernel/network/ip/ip.h)) 23c⟩
⟨macro ipcmp((kernel/network/ip/ip.h)) 23d⟩

extern ipaddr IPv4bcast;
extern ipaddr IPnoaddr;
extern ipaddr v4prefix;
extern ipaddr IPallbits;
//extern ipaddr IPv4bcastobs;
//extern ipaddr IPv4allsys;
//extern ipaddr IPv4allrouter;

⟨constant NOW 234a⟩

/*
 *  media
 */
//extern Medium ethermedium;
//extern Medium nullmedium;
//extern Medium pktmedium;

/*
 *  ipifc.c
 */
extern Medium*  ipfindmedium(char *name);
extern void addipmedium(Medium *med);
extern int  ipforme(Fs*, uchar *addr);
extern int  iptentative(Fs*, uchar *addr);
//extern int  ipisbm(uchar *);
extern int  ipismulticast(uchar *);
extern Ipifc* findipifc(Fs*, uchar *remote, int type);
extern void findlocalip(Fs*, uchar *local, uchar *remote);
extern int  ipv4local(Ipifc *ifc, uchar *addr);
extern int  ipv6local(Ipifc *ifc, uchar *addr);
```

```
extern int  ipv6anylocal(Ipifc *ifc, uchar *addr);
extern Iplifc*  iplocalonifc(Ipifc *ifc, uchar *ip);
extern int  ipproxyifc(Fs *f, Ipifc *ifc, uchar *ip);
extern int  ipismulticast(uchar *ip);
//extern int  ipisbooting(void);
//extern int  ipifccheckin(Ipifc *ifc, Medium *med);
//extern void ipifccheckout(Ipifc *ifc);
//extern int  ipifcgrab(Ipifc *ifc);
extern void ipifcaddroute(Fs*, int, uchar*, uchar*, uchar*, int);
extern void ipifcremroute(Fs*, int, uchar*, uchar*);
extern void ipifcremmulti(Conv *c, uchar *ma, uchar *ia);
extern void ipifcaddmulti(Conv *c, uchar *ma, uchar *ia);
//extern char*  ipifcrem(Ipifc *ifc, char **argv, int argc);
//extern char*  ipifcadd(Ipifc *ifc, char **argv, int argc, int tentative, Iplifc *lifcp);
extern long ipselftabread(Fs*, char *a, ulong offset, int n);
//extern char*  ipifcadd6(Ipifc *ifc, char**argv, int argc);

/*
 *  ip.c
 */
extern void iprouting(Fs*, int);
extern void icmpnoconv(Fs*, Block*);
extern void icmpcantfrag(Fs*, Block*, int);
extern void icmpttlexceeded(Fs*, uchar*, Block*);
extern ushort ipcsum(uchar*);
extern void ipiput4(Fs*, Ipifc*, Block*);
extern void ipiput6(Fs*, Ipifc*, Block*);
extern int  ipoput4(Fs*, Block*, int, int, int, Conv*);
extern int  ipoput6(Fs*, Block*, int, int, int, Conv*);
extern int  ipstats(Fs*, char*, int);
extern ushort ptclbsum(uchar*, int);
extern ushort ptclcsum(Block*, int, int);
extern void ip_init(Fs*);
//extern void update_mtucache(uchar*, ulong);
//extern ulong  restrict_mtu(uchar*, ulong);

/*
 * bootp.c
 */
extern int  bootpread(char*, ulong, int);

/*
 * chandial.c
 */
extern Chan*  chandial(char*, char*, char*, Chan**);

/*
 *  global to all of the stack
 */
//extern void (*igmpreportfn)(Ipifc*, uchar*);
```

Uses `Arp` 106b, `Arpent` 234b, `Conv` 234b, `Fragment4` 234b, `Fragment6` 234b, `Fs` 234b, `IP` 234b, `IPaux` 234b, `Ip4hdr` 234b, `Iphash` 234b, `Ipht` 234b, `Ipifc` 234b, `Iplifc` 234b, `Iplink` 234b, `Ipmulti` 234b, `Ipself` 126c, `Ipselftab` 126a, `Medium` 234b, `Netlog` 188b, `Proto` 234b, `Route` 234b, `RouteTree` 234b, `Routewalk` 234b, `V4route` 234b, `V6route` 234b, and `v6params` 234b.

I.4.2 kernel/network/ip/ip.c

⟨*macro* BKFG 238⟩≡ (239a)

```
/*
 * This sleazy macro relies on the media header size being
```

```
 * larger than sizeof(Ipfrag). ipreassemble checks this is true
 */
#define BKFG(xp)    ((Ipfrag*)((xp)->base))
```

⟨kernel/network/ip/ip.c 239a⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"

#include    "ip.h"

⟨macro BLKIPVER 39c⟩

⟨global statnames 31b⟩

⟨macro BLKIP 39d⟩
⟨macro BKFG 238⟩

ushort      ipcsum(uchar*);
Block*      ip4reassemble(IP*, int, Block*, Ip4hdr*);
void        ipfragfree4(IP*, Fragment4*);
Fragment4*  ipfragallo4(IP*);

⟨function ip_init_6 502b⟩

⟨function initfrag 42b⟩

⟨function ip_init 42a⟩

⟨function iprouting 80a⟩

⟨function ipoput4 93⟩

⟨function ipiput4 98⟩

⟨function ipstats 77b⟩

⟨function ip4reassemble 100c⟩

⟨function ipfragfree4 92a⟩

⟨function ipfragallo4 92b⟩

⟨function ipcsum 91⟩
```

I.5 kernel/network/

I.5.1 kernel/network/portfns_network.h

⟨kernel/network/portfns_network.h 239b⟩≡

```
void   hnputl(void*, uint);
void   hnputs(void*, ushort);
uint   nhgetl(void*);
ushort nhgets(void*);
```

```
//void    hnputv(void*, uvlong);
//uvlong    nhgetv(void*);
```

I.5.2 kernel/network/netif.h

⟨*enum* _anon_ (kernel/network/netif.h) 240a⟩≡ (241a)

```
enum
{
  Nmaxaddr= 64,
  Nmhash=   31,

  Ncloneqid=  1,
  Naddrqid,
  N2ndqid,
  N3rdqid,
  Ndataqid,
  Nctlqid,
  Nstatqid,
  Ntypeqid,
  Nifstatqid,
  Nmtuqid,
};
```

⟨*macro* NETTYPE 240b⟩≡ (241a)

```
/*
 *  Macros to manage Qid's used for multiplexed devices
 */
#define NETTYPE(x)  (((ulong)x)&0x1f)
```

⟨*macro* NETID 240c⟩≡ (241a)

```
#define NETID(x)  ((((ulong)x))>>5)
```

⟨*macro* NETQID 240d⟩≡ (241a)

```
#define NETQID(i,t) ((((ulong)i)<<5)|(t))
```

⟨*struct* Netaddr 240e⟩≡ (241a)

```
/*
 *  a network address
 */
struct Netaddr
{
  Netaddr *next;    /* allocation chain */
  Netaddr *hnext;
  uchar addr[Nmaxaddr];
  int ref;
};
```

Uses Nmaxaddr 240a.

⟨*enum* _anon_ (kernel/network/netif.h)2 240f⟩≡ (241a)

```
/*
 *  Ethernet specific
 */
enum
{
  ⟨constant Eaddrlen 27d⟩
  ETHERMINTU =  60,   /* minimum transmit size */
  ETHERMAXTU =  1514,   /* maximum transmit size */
  ETHERHDRSIZE =  14,   /* size of an ethernet header */
```

```
  /* ethernet packet types */
  ETARP   = 0x0806,
  ETIP4   = 0x0800,

  ETIP6   = 0x86DD,
};
```

⟨kernel/network/netif.h 241a⟩≡

```
// todo: split in portdat_network.h? and portfns_network.h

typedef struct Etherpkt Etherpkt;
typedef struct Netaddr  Netaddr;
typedef struct Netfile  Netfile;
typedef struct Netif  Netif;

⟨enum _anon_ (kernel/network/netif.h) 240a⟩

⟨macro NETTYPE 240b⟩
⟨macro NETID 240c⟩
⟨macro NETQID 240d⟩

⟨struct Netfile 37a⟩

⟨struct Netaddr 240e⟩

⟨struct Netif (kernel) 35f⟩

void  netifinit(Netif*, char*, int, ulong);
Walkqid*  netifwalk(Netif*, Chan*, Chan*, char **, int);
Chan* netifopen(Netif*, Chan*, int);
void  netifclose(Netif*, Chan*);
long  netifread(Netif*, Chan*, void*, long, ulong);
Block*  netifbread(Netif*, Chan*, long, ulong);
long  netifwrite(Netif*, Chan*, void*, long);
int netifwstat(Netif*, Chan*, uchar*, int);
int netifstat(Netif*, Chan*, uchar*, int);
int activemulti(Netif*, uchar*, int);

⟨enum _anon_ (kernel/network/netif.h)2 240f⟩

⟨typedef eaddr 27e⟩

⟨struct Etherpkt 86b⟩
```

Uses Etherpkt 86b, Netaddr 240e, Netfile 37a, and Netif 35f.

I.5.3 kernel/network/netif.c

⟨*function* netifinit 241b⟩≡ (254)

```
/*
 *  set up a new network interface
 */
void
netifinit(Netif *nif, char *name, int nfile, ulong limit)
{
    strncpy(nif->name, name, KNAMELEN-1);
```

```
    nif->name[KNAMELEN-1] = 0;
    nif->nfile = nfile;
    nif->f = xalloc(nfile*sizeof(Netfile*));
    if (nif->f == nil)
        panic("netifinit: no memory");
    memset(nif->f, 0, nfile*sizeof(Netfile*));
    nif->limit = limit;
}
```

⟨*function* netifgen 242⟩≡ (254)

```
/*
 *  generate a 3 level directory
 */
static int
netifgen(Chan *c, char*, Dirtab *vp, int, int i, Dir *dp)
{
    Qid q;
    Netif *nif = (Netif*)vp;
    Netfile *f;
    int t;
    int perm;
    char *o;

    q.type = QTFILE;
    q.vers = 0;

    /* top level directory contains the name of the network */
    if(c->qid.path == 0){
        switch(i){
        case DEVDOTDOT:
            q.path = 0;
            q.type = QTDIR;
            devdir(c, q, ".", 0, eve, 0555, dp);
            break;
        case 0:
            q.path = N2ndqid;
            q.type = QTDIR;
            strcpy(up->genbuf, nif->name);
            devdir(c, q, up->genbuf, 0, eve, 0555, dp);
            break;
        default:
            return -1;
        }
        return 1;
    }

    /* second level contains clone plus all the conversations */
    t = NETTYPE(c->qid.path);
    if(t == N2ndqid || t == Ncloneqid || t == Naddrqid){
        switch(i) {
        case DEVDOTDOT:
            q.type = QTDIR;
            q.path = 0;
            devdir(c, q, ".", 0, eve, DMDIR|0555, dp);
            break;
        case 0:
            q.path = Ncloneqid;
            devdir(c, q, "clone", 0, eve, 0666, dp);
            break;
        case 1:
```

```
        q.path = Naddrqid;
        devdir(c, q, "addr", 0, eve, 0666, dp);
        break;
    case 2:
        q.path = Nstatqid;
        devdir(c, q, "stats", 0, eve, 0444, dp);
        break;
    case 3:
        q.path = Nifstatqid;
        devdir(c, q, "ifstats", 0, eve, 0444, dp);
        break;
    default:
        i -= 4;
        if(i >= nif->nfile)
            return -1;
        if(nif->f[i] == 0)
            return 0;
        q.type = QTDIR;
        q.path = NETQID(i, N3rdqid);
        snprint(up->genbuf, sizeof up->genbuf, "%d", i);
        devdir(c, q, up->genbuf, 0, eve, DMDIR|0555, dp);
        break;
    }
    return 1;
}

/* third level */
f = nif->f[NETID(c->qid.path)];
if(f == 0)
    return 0;
if(*f->owner){
    o = f->owner;
    perm = f->mode;
} else {
    o = eve;
    perm = 0666;
}
switch(i){
case DEVDOTDOT:
    q.type = QTDIR;
    q.path = N2ndqid;
    strcpy(up->genbuf, nif->name);
    devdir(c, q, up->genbuf, 0, eve, DMDIR|0555, dp);
    break;
case 0:
    q.path = NETQID(NETID(c->qid.path), Ndataqid);
    devdir(c, q, "data", 0, o, perm, dp);
    break;
case 1:
    q.path = NETQID(NETID(c->qid.path), Nctlqid);
    devdir(c, q, "ctl", 0, o, perm, dp);
    break;
case 2:
    q.path = NETQID(NETID(c->qid.path), Nstatqid);
    devdir(c, q, "stats", 0, eve, 0444, dp);
    break;
case 3:
    q.path = NETQID(NETID(c->qid.path), Ntypeqid);
    devdir(c, q, "type", 0, eve, 0444, dp);
    break;
```

```
    case 4:
        q.path = NETQID(NETID(c->qid.path), Nifstatqid);
        devdir(c, q, "ifstats", 0, eve, 0444, dp);
        break;
    default:
        return -1;
    }
    return 1;
}
```

Uses N2ndqid 240a, N3rdqid 240a, NETID 240c, NETQID 240d, NETTYPE 240b, Naddrqid 240a, Ncloneqid 240a, Nctlqid 240a, Ndataqid 240a, Nifstatqid 240a, Nstatqid 240a, and Ntypeqid 240a.

⟨*function* netifwalk 244a⟩≡ (254)

```
Walkqid*
netifwalk(Netif *nif, Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, (Dirtab *)nif, 0, netifgen);
}
```

Uses netifgen() 242.

⟨*function* netifopen 244b⟩≡ (254)

```
Chan*
netifopen(Netif *nif, Chan *c, int omode)
{
    int id;
    Netfile *f;

    id = 0;
    if(c->qid.type & QTDIR){
        if(omode != OREAD)
            error(Eperm);
    } else {
        switch(NETTYPE(c->qid.path)){
        case Ndataqid:
        case Nctlqid:
            id = NETID(c->qid.path);
            openfile(nif, id);
            break;
        case Ncloneqid:
            id = openfile(nif, -1);
            c->qid.path = NETQID(id, Nctlqid);
            break;
        default:
            if(omode != OREAD)
                error(Ebadarg);
        }
        switch(NETTYPE(c->qid.path)){
        case Ndataqid:
        case Nctlqid:
            f = nif->f[id];
            if(netown(f, up->user, omode&7) < 0)
                error(Eperm);
            break;
        }
    }
    c->mode = openmode(omode);
    c->flag |= COPEN;
    c->offset = 0;
    c->iounit = qiomaxatomic;
    return c;
```

```
    }
```

Uses NETID 240c, NETQID 240d, NETTYPE 240b, Ncloneqid 240a, Nctlqid 240a, Ndataqid 240a, netown() 249b, and openfile() 250.

⟨*function* netifread 245⟩≡ (254)

```
    long
    netifread(Netif *nif, Chan *c, void *a, long n, ulong offset)
    {
        int i, j;
        Netfile *f;
        char *p;

        if(c->qid.type&QTDIR)
            return devdirread(c, a, n, (Dirtab*)nif, 0, netifgen);

        switch(NETTYPE(c->qid.path)){
        case Ndataqid:
            f = nif->f[NETID(c->qid.path)];
            return qread(f->in, a, n);
        case Nctlqid:
            return readnum(offset, a, n, NETID(c->qid.path), NUMSIZE);
        case Nstatqid:
            p = malloc(READSTR);
            if(p == nil)
                error(Enomem);
            j = snprint(p, READSTR, "in: %llud\n", nif->inpackets);
            j += snprint(p+j, READSTR-j, "link: %d\n", nif->link);
            j += snprint(p+j, READSTR-j, "out: %llud\n", nif->outpackets);
            j += snprint(p+j, READSTR-j, "crc errs: %d\n", nif->crcs);
            j += snprint(p+j, READSTR-j, "overflows: %d\n", nif->overflows);
            j += snprint(p+j, READSTR-j, "soft overflows: %d\n", nif->soverflows);
            j += snprint(p+j, READSTR-j, "framing errs: %d\n", nif->frames);
            j += snprint(p+j, READSTR-j, "buffer errs: %d\n", nif->buffs);
            j += snprint(p+j, READSTR-j, "output errs: %d\n", nif->oerrs);
            j += snprint(p+j, READSTR-j, "prom: %d\n", nif->prom);
            j += snprint(p+j, READSTR-j, "mbps: %d\n", nif->mbps);
            j += snprint(p+j, READSTR-j, "addr: ");
            for(i = 0; i < nif->alen; i++)
                j += snprint(p+j, READSTR-j, "%2.2ux", nif->addr[i]);
            snprint(p+j, READSTR-j, "\n");
            n = readstr(offset, a, n, p);
            free(p);
            return n;
        case Naddrqid:
            p = malloc(READSTR);
            if(p == nil)
                error(Enomem);
            j = 0;
            for(i = 0; i < nif->alen; i++)
                j += snprint(p+j, READSTR-j, "%2.2ux", nif->addr[i]);
            n = readstr(offset, a, n, p);
            free(p);
            return n;
        case Ntypeqid:
            f = nif->f[NETID(c->qid.path)];
            return readnum(offset, a, n, f->type, NUMSIZE);
        case Nifstatqid:
            return 0;
        }
        error(Ebadarg);
        return -1;  /* not reached */
```

```
}
```

Uses NETID 240c, NETTYPE 240b, Naddrqid 240a, Nctlqid 240a, Ndataqid 240a, Nifstatqid 240a, Nstatqid 240a, Ntypeqid 240a, netifgen() 242, and readstr().

⟨*function* netifbread 246a⟩≡ (254)

```
Block*
netifbread(Netif *nif, Chan *c, long n, ulong offset)
{
    if((c->qid.type & QTDIR) || NETTYPE(c->qid.path) != Ndataqid)
        return devbread(c, n, offset);

    return qbread(nif->f[NETID(c->qid.path)]->in, n);
}
```

Uses NETID 240c, NETTYPE 240b, and Ndataqid 240a.

⟨*function* typeinuse 246b⟩≡ (254)

```
/*
 *  make sure this type isn't already in use on this device
 */
static int
typeinuse(Netif *nif, int type)
{
    Netfile *f, **fp, **efp;

    if(type <= 0)
        return 0;

    efp = &nif->f[nif->nfile];
    for(fp = nif->f; fp < efp; fp++){
        f = *fp;
        if(f == 0)
            continue;
        if(f->type == type)
            return 1;
    }
    return 0;
}
```

⟨*function* netifwrite 246c⟩≡ (254)

```
/*
 *  the devxxx.c that calls us handles writing data, it knows best
 */
long
netifwrite(Netif *nif, Chan *c, void *a, long n)
{
    Netfile *f;
    int type;
    char *p, buf[64];
    uchar binaddr[Nmaxaddr];

    if(NETTYPE(c->qid.path) != Nctlqid)
        error(Eperm);

    if(n >= sizeof(buf))
        n = sizeof(buf)-1;
    memmove(buf, a, n);
    buf[n] = 0;

    if(waserror()){
        qunlock(nif);
```

```
        nexterror();
    }

    qlock(nif);
    f = nif->f[NETID(c->qid.path)];
    if((p = matchtoken(buf, "connect")) != 0){
        type = atoi(p);
        if(typeinuse(nif, type))
            error(Einuse);
        f->type = type;
        if(f->type < 0)
            nif->all++;
    } else if(matchtoken(buf, "promiscuous")){
        if(f->prom == 0){
            if(nif->prom == 0 && nif->promiscuous != nil)
                nif->promiscuous(nif->arg, 1);
            f->prom = 1;
            nif->prom++;
        }
    } else if((p = matchtoken(buf, "scanbs")) != 0){
        /* scan for base stations */
        if(f->scan == 0){
            type = atoi(p);
            if(type < 5)
                type = 5;
            if(nif->scanbs != nil)
                nif->scanbs(nif->arg, type);
            f->scan = type;
            nif->scan++;
        }
    } else if(matchtoken(buf, "bridge")){
        f->bridge = 1;
    } else if(matchtoken(buf, "headersonly")){
        f->headersonly = 1;
    } else if((p = matchtoken(buf, "addmulti")) != 0){
        if(parseaddr(binaddr, p, nif->alen) < 0)
            error("bad address");
        p = netmulti(nif, f, binaddr, 1);
        if(p)
            error(p);
    } else if((p = matchtoken(buf, "remmulti")) != 0){
        if(parseaddr(binaddr, p, nif->alen) < 0)
            error("bad address");
        p = netmulti(nif, f, binaddr, 0);
        if(p)
            error(p);
    } else
        n = -1;
    qunlock(nif);
    poperror();
    return n;
}
```

Uses `NETID` 240c, `NETTYPE` 240b, `Nctlqid` 240a, `Nmaxaddr` 240a, `matchtoken()` 251a, `netmulti()` 253, `parseaddr()` 252d, and `typeinuse()` 246b.

⟨*function* `netifwstat` 247⟩≡ (254)

```
int
netifwstat(Netif *nif, Chan *c, uchar *db, int n)
{
    Dir *dir;
```

```
    Netfile *f;
    int m;

    f = nif->f[NETID(c->qid.path)];
    if(f == 0)
        error(Enonexist);

    if(netown(f, up->user, OWRITE) < 0)
        error(Eperm);

    dir = smalloc(sizeof(Dir)+n);
    m = convM2D(db, n, &dir[0], (char*)&dir[1]);
    if(m == 0){
        free(dir);
        error(Eshortstat);
    }
    if(!emptystr(dir[0].uid))
        strncpy(f->owner, dir[0].uid, KNAMELEN);
    if(dir[0].mode != ~0UL)
        f->mode = dir[0].mode;
    free(dir);
    return m;
}
```

Uses NETID 240c and netown() 249b.

⟨*function* netifstat 248a⟩≡ (254)

```
int
netifstat(Netif *nif, Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, (Dirtab *)nif, 0, netifgen);
}
```

Uses netifgen() 242.

⟨*function* netifclose 248b⟩≡ (254)

```
void
netifclose(Netif *nif, Chan *c)
{
    Netfile *f;
    int t;
    Netaddr *ap;

    if((c->flag & COPEN) == 0)
        return;

    t = NETTYPE(c->qid.path);
    if(t != Ndataqid && t != Nctlqid)
        return;

    f = nif->f[NETID(c->qid.path)];
    qlock(f);
    if(--(f->inuse) == 0){
        if(f->prom){
            qlock(nif);
            if(--(nif->prom) == 0 && nif->promiscuous != nil)
                nif->promiscuous(nif->arg, 0);
            qunlock(nif);
            f->prom = 0;
        }
        if(f->scan){
            qlock(nif);
```

```
            if(--(nif->scan) == 0 && nif->scanbs != nil)
                nif->scanbs(nif->arg, 0);
            qunlock(nif);
            f->prom = 0;
            f->scan = 0;
        }
        if(f->nmaddr){
            qlock(nif);
            t = 0;
            for(ap = nif->maddr; ap; ap = ap->next){
                if(f->maddr[t/8] & (1<<(t%8)))
                    netmulti(nif, f, ap->addr, 0);
            }
            qunlock(nif);
            f->nmaddr = 0;
        }
        if(f->type < 0){
            qlock(nif);
            --(nif->all);
            qunlock(nif);
        }
        f->owner[0] = 0;
        f->type = 0;
        f->bridge = 0;
        f->headersonly = 0;
        qclose(f->in);
    }
    qunlock(f);
}
```

Uses `NETID` 240c, `NETTYPE` 240b, `Nctlqid` 240a, `Ndataqid` 240a, and `netmulti()` 253.

⟨*global* `netlock` 249a⟩≡ (254)

```
Lock netlock;
```

⟨*function* `netown` 249b⟩≡ (254)

```
static int
netown(Netfile *p, char *o, int omode)
{
    static int access[] = { 0400, 0200, 0600, 0100 };
    int mode;
    int t;

    lock(&netlock);
    if(*p->owner){
        if(strncmp(o, p->owner, KNAMELEN) == 0) /* User */
            mode = p->mode;
        else if(strncmp(o, eve, KNAMELEN) == 0) /* Bootes is group */
            mode = p->mode<<3;
        else
            mode = p->mode<<6;      /* Other */

        t = access[omode&3];
        if((t & mode) == t){
            unlock(&netlock);
            return 0;
        } else {
            unlock(&netlock);
            return -1;
        }
    }
```

```
    strncpy(p->owner, o, KNAMELEN);
    p->mode = 0660;
    unlock(&netlock);
    return 0;
}
```

Uses netlock 249a.

⟨*function* openfile 250⟩≡ (254)

```
/*
 *  Increment the reference count of a network device.
 *  If id < 0, return an unused ether device.
 */
static int
openfile(Netif *nif, int id)
{
    Netfile *f, **fp, **efp;

    if(id >= 0){
        f = nif->f[id];
        if(f == 0)
            error(Enodev);
        qlock(f);
        qreopen(f->in);
        f->inuse++;
        qunlock(f);
        return id;
    }

    qlock(nif);
    if(waserror()){
        qunlock(nif);
        nexterror();
    }
    efp = &nif->f[nif->nfile];
    for(fp = nif->f; fp < efp; fp++){
        f = *fp;
        if(f == 0){
            f = malloc(sizeof(Netfile));
            if(f == 0)
                exhausted("memory");
            f->in = qopen(nif->limit, Qmsg, 0, 0);
            if(f->in == nil){
                free(f);
                exhausted("memory");
            }
            *fp = f;
            qlock(f);
        } else {
            qlock(f);
            if(f->inuse){
                qunlock(f);
                continue;
            }
        }
        f->inuse = 1;
        qreopen(f->in);
        netown(f, up->user, 0);
        qunlock(f);
        qunlock(nif);
        poperror();
```

```
        return fp - nif->f;
    }
    error(Enodev);
    return -1;  /* not reached */
}
```

Uses netown() 249b.

⟨*function* matchtoken 251a⟩≡ (254)

```
/*
 *  look for a token starting a string,
 *  return a pointer to first non-space char after it
 */
static char*
matchtoken(char *p, char *token)
{
    int n;

    n = strlen(token);
    if(strncmp(p, token, n))
        return 0;
    p += n;
    if(*p == 0)
        return p;
    if(*p != ' ' && *p != '\t' && *p != '\n')
        return 0;
    while(*p == ' ' || *p == '\t' || *p == '\n')
        p++;
    return p;
}
```

⟨*function* hnputl*(*(kernel/network/netif.c)*)* 251b⟩≡ (254)

```
void
hnputl(void *p, uint v)
{
    uchar *a;

    a = p;
    a[0] = v>>24;
    a[1] = v>>16;
    a[2] = v>>8;
    a[3] = v;
}
```

⟨*function* hnputs*(*(kernel/network/netif.c)*)* 251c⟩≡ (254)

```
void
hnputs(void *p, ushort v)
{
    uchar *a;

    a = p;
    a[0] = v>>8;
    a[1] = v;
}
```

⟨*function* nhgetl*(*(kernel/network/netif.c)*)* 251d⟩≡ (254)

```
uint
nhgetl(void *p)
{
    uchar *a;
```

```
    a = p;
    return (a[0]<<24)|(a[1]<<16)|(a[2]<<8)|(a[3]<<0);
}
```

⟨*function* nhgets*(*(kernel/network/netif.c)*)* 252a⟩≡ (254)

```
ushort
nhgets(void *p)
{
    uchar *a;

    a = p;
    return (a[0]<<8)|(a[1]<<0);
}
```

⟨*function* hash 252b⟩≡ (254)

```
static ulong
hash(uchar *a, int len)
{
    ulong sum = 0;

    while(len-- > 0)
        sum = (sum << 1) + *a++;
    return sum%Nmhash;
}
```

Uses Nmhash 240a.

⟨*function* activemulti 252c⟩≡ (254)

```
int
activemulti(Netif *nif, uchar *addr, int alen)
{
    Netaddr *hp;

    for(hp = nif->mhash[hash(addr, alen)]; hp; hp = hp->hnext)
        if(memcmp(addr, hp->addr, alen) == 0){
            if(hp->ref)
                return 1;
            else
                break;
        }
    return 0;
}
```

Uses hash() 252b.

⟨*function* parseaddr 252d⟩≡ (254)

```
static int
parseaddr(uchar *to, char *from, int alen)
{
    char nip[4];
    char *p;
    int i;

    p = from;
    for(i = 0; i < alen; i++){
        if(*p == 0)
            return -1;
        nip[0] = *p++;
        if(*p == 0)
            return -1;
        nip[1] = *p++;
        nip[2] = 0;
```

```
        to[i] = strtoul(nip, 0, 16);
        if(*p == ':')
            p++;
    }
    return 0;
}
```

⟨*function* netmulti 253⟩≡ (254)

```
/*
 *  keep track of multicast addresses
 */
static char*
netmulti(Netif *nif, Netfile *f, uchar *addr, int add)
{
    Netaddr **l, *ap;
    int i;
    ulong h;

    if(nif->multicast == nil)
        return "interface does not support multicast";

    l = &nif->maddr;
    i = 0;
    for(ap = *l; ap; ap = *l){
        if(memcmp(addr, ap->addr, nif->alen) == 0)
            break;
        i++;
        l = &ap->next;
    }

    if(add){
        if(ap == 0){
            *l = ap = smalloc(sizeof(*ap));
            memmove(ap->addr, addr, nif->alen);
            ap->next = 0;
            ap->ref = 1;
            h = hash(addr, nif->alen);
            ap->hnext = nif->mhash[h];
            nif->mhash[h] = ap;
        } else {
            ap->ref++;
        }
        if(ap->ref == 1){
            nif->nmaddr++;
            nif->multicast(nif->arg, addr, 1);
        }
        if(i < 8*sizeof(f->maddr)){
            if((f->maddr[i/8] & (1<<(i%8))) == 0)
                f->nmaddr++;
            f->maddr[i/8] |= 1<<(i%8);
        }
    } else {
        if(ap == 0 || ap->ref == 0)
            return 0;
        ap->ref--;
        if(ap->ref == 0){
            nif->nmaddr--;
            nif->multicast(nif->arg, addr, 0);
        }
        if(i < 8*sizeof(f->maddr)){
```

```
            if((f->maddr[i/8] & (1<<(i%8))) != 0)
                f->nmaddr--;
            f->maddr[i/8] &= ~(1<<(i%8));
        }
    }
    return 0;
}
```

Uses hash() 252b.

⟨kernel/network/netif.c 254⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"
#include    "../port/netif.h"

static int netown(Netfile*, char*, int);
static int openfile(Netif*, int);
static char* matchtoken(char*, char*);
static char* netmulti(Netif*, Netfile*, uchar*, int);
static int parseaddr(uchar*, char*, int);

⟨function netifinit 241b⟩

⟨function netifgen 242⟩

⟨function netifwalk 244a⟩

⟨function netifopen 244b⟩

⟨function netifread 245⟩

⟨function netifbread 246a⟩

⟨function typeinuse 246b⟩

⟨function netifwrite 246c⟩

⟨function netifwstat 247⟩

⟨function netifstat 248a⟩

⟨function netifclose 248b⟩

⟨global netlock 249a⟩

⟨function netown 249b⟩

⟨function openfile 250⟩

⟨function matchtoken 251a⟩

⟨function hnputl((kernel/network/netif.c)) 251b⟩

⟨function hnputs((kernel/network/netif.c)) 251c⟩

⟨function nhgetl((kernel/network/netif.c)) 251d⟩
```

⟨*function* nhgets *(*(kernel/network/netif.c)*)* 252a⟩

⟨*function* hash 252b⟩

⟨*function* activemulti 252c⟩

⟨*function* parseaddr 252d⟩

⟨*function* netmulti 253⟩

I.6 kernel/network/386/

I.6.1 kernel/network/etherif.h

⟨*enum* _anon_ (kernel/network/etherif.h) 255a⟩≡ (255d)

```
enum {
  ⟨constant MaxEther 28d⟩
  Ntypes     = 8,
};
```

⟨*macro* NEXT 255b⟩≡ (255d)

```
#define NEXT(x, l)  (((x)+1)%(l))
```

⟨*macro* PREV 255c⟩≡ (255d)

```
#define PREV(x, l)  (((x) == 0) ? (l)-1: (x)-1)
```

⟨kernel/network/etherif.h 255d⟩≡

```
⟨enum _anon_ (kernel/network/etherif.h) 255a⟩

typedef struct Ether Ether;
⟨struct Ether (kernel) 28e⟩

extern Block* etheriq(Ether*, Block*, int);
extern void addethercard(char*, int(*)(Ether*));
extern ulong ethercrc(uchar*, int);
//extern int parseether(uchar*, char*);

⟨macro NEXT 255b⟩
⟨macro PREV 255c⟩
```

Uses Ether 28e.

I.6.2 kernel/network/386/devether.c

⟨*function* etherwalk 255e⟩≡ (264b)

```
static Walkqid*
etherwalk(Chan* chan, Chan* nchan, char** name, int nname)
{
    return netifwalk(etherxx[chan->dev], chan, nchan, name, nname);
}
```

Uses etherxx-795 28c and netifwalk() 244a.

⟨*function* etherstat 255f⟩≡ (264b)

```
static int
etherstat(Chan* chan, uchar* dp, int n)
{
    return netifstat(etherxx[chan->dev], chan, dp, n);
}
```

Uses etherxx-795 28c and netifstat() 248a.

⟨*function* etheropen 256a⟩≡ (264b)
```
static Chan*
etheropen(Chan* chan, int omode)
{
    return netifopen(etherxx[chan->dev], chan, omode);
}
```
Uses etherxx-795 28c and netifopen() 244b.

⟨*function* ethercreate 256b⟩≡ (264b)
```
static void
ethercreate(Chan*, char*, int, ulong)
{
}
```

⟨*function* etherclose 256c⟩≡ (264b)
```
static void
etherclose(Chan* chan)
{
    netifclose(etherxx[chan->dev], chan);
}
```
Uses etherxx-795 28c and netifclose() 248b.

⟨*function* etherread 256d⟩≡ (264b)
```
static long
etherread(Chan* chan, void* buf, long n, vlong off)
{
    Ether *ether;
    ulong offset = off;

    ether = etherxx[chan->dev];
    if((chan->qid.type & QTDIR) == 0 && ether->ifstat){
        /*
         * With some controllers it is necessary to reach
         * into the chip to extract statistics.
         */
        if(NETTYPE(chan->qid.path) == Nifstatqid)
            return ether->ifstat(ether, buf, n, offset);
        else if(NETTYPE(chan->qid.path) == Nstatqid)
            ether->ifstat(ether, buf, 0, offset);
    }

    return netifread(ether, chan, buf, n, offset);
}
```
Uses NETTYPE 240b, Nifstatqid 240a, Nstatqid 240a, etherxx-795 28c, and netifread() 245.

⟨*function* etherbread 256e⟩≡ (264b)
```
static Block*
etherbread(Chan* chan, long n, ulong offset)
{
    return netifbread(etherxx[chan->dev], chan, n, offset);
}
```
Uses etherxx-795 28c and netifbread() 246a.

⟨*function* etherwstat 256f⟩≡ (264b)
```
static int
etherwstat(Chan* chan, uchar* dp, int n)
{
    return netifwstat(etherxx[chan->dev], chan, dp, n);
}
```
Uses etherxx-795 28c and netifwstat() 247.

⟨*function* etherrtrace 257a⟩≡ (264b)

```
static void
etherrtrace(Netfile* f, Etherpkt* pkt, int len)
{
    int i, n;
    Block *bp;

    if(qwindow(f->in) <= 0)
        return;
    if(len > 58)
        n = 58;
    else
        n = len;
    bp = iallocb(64);
    if(bp == nil)
        return;
    memmove(bp->wp, pkt->d, n);
    i = TK2MS(CPUS(0)->ticks);
    bp->wp[58] = len>>8;
    bp->wp[59] = len;
    bp->wp[60] = i>>24;
    bp->wp[61] = i>>16;
    bp->wp[62] = i>>8;
    bp->wp[63] = i;
    bp->wp += 64;
    qpass(f->in, bp);
}
```

⟨*function* etheriq 257b⟩≡ (264b)

```
Block*
etheriq(Ether* ether, Block* bp, int fromwire)
{
    Etherpkt *pkt;
    ushort type;
    int len, multi, tome, fromme;
    Netfile **ep, *f, **fp, *fx;
    Block *xbp;

    ether->inpackets++;

    pkt = (Etherpkt*)bp->rp;
    len = BLEN(bp);
    type = (pkt->type[0]<<8)|pkt->type[1];
    fx = 0;
    ep = &ether->f[Ntypes];

    multi = pkt->d[0] & 1;
    /* check for valid multicast addresses */
    if(multi && memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) != 0 && ether->prom == 0){
        if(!activemulti(ether, pkt->d, sizeof(pkt->d))){
            if(fromwire){
                freeb(bp);
                bp = 0;
            }
            return bp;
        }
    }

    /* is it for me? */
    tome = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
```

```
    fromme = memcmp(pkt->s, ether->ea, sizeof(pkt->s)) == 0;

    /*
     * Multiplex the packet to all the connections which want it.
     * If the packet is not to be used subsequently (fromwire != 0),
     * attempt to simply pass it into one of the connections, thereby
     * saving a copy of the data (usual case hopefully).
     */
    for(fp = ether->f; fp < ep; fp++){
        if(f = *fp)
        if(f->type == type || f->type < 0)
        if(tome || multi || f->prom){
            /* Don't want to hear bridged packets */
            if(f->bridge && !fromwire && !fromme)
                continue;
            if(!f->headersonly){
                if(fromwire && fx == 0)
                    fx = f;
                else if(xbp = iallocb(len)){
                    memmove(xbp->wp, pkt, len);
                    xbp->wp += len;
                    if(qpass(f->in, xbp) < 0){
                        // print("soverflow for f->in\n");
                        ether->soverflows++;
                    }
                }
                else{
                    // print("soverflow iallocb\n");
                    ether->soverflows++;
                }
            }
            else
                etherrtrace(f, pkt, len);
        }
    }

    if(fx){
        if(qpass(fx->in, bp) < 0){
            // print("soverflow for fx->in\n");
            ether->soverflows++;
        }
        return 0;
    }
    if(fromwire){
        freeb(bp);
        return 0;
    }

    return bp;
}
```
Uses Ntypes 255a and activemulti() 252c.

⟨*function* etheroq 258⟩≡ (264b)
```
static int
etheroq(Ether* ether, Block* bp)
{
    int len, loopback, s;
    Etherpkt *pkt;

    ether->outpackets++;
```

```
    /*
     * Check if the packet has to be placed back onto the input queue,
     * i.e. if it's a loopback or broadcast packet or the interface is
     * in promiscuous mode.
     * If it's a loopback packet indicate to etheriq that the data isn't
     * needed and return, etheriq will pass-on or free the block.
     * To enable bridging to work, only packets that were originated
     * by this interface are fed back.
     */
    pkt = (Etherpkt*)bp->rp;
    len = BLEN(bp);
    loopback = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
    if(loopback || memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) == 0 || ether->prom){
        s = arch_splhi();
        etheriq(ether, bp, 0);
        arch_splx(s);
    }

    if(!loopback){
        if(qfull(ether->oq))
            print("etheroq: WARNING: ether->oq full!\n");
        qbwrite(ether->oq, bp);
        if(ether->transmit != nil)
            ether->transmit(ether);
    } else
        freeb(bp);

    return len;
}
```

⟨*function* etherwrite 259⟩≡ (264b)

```
static long
etherwrite(Chan* chan, void* buf, long n, vlong)
{
    Ether *ether;
    Block *bp;
    int nn, onoff;
    Cmdbuf *cb;

    ether = etherxx[chan->dev];
    if(NETTYPE(chan->qid.path) != Ndataqid) {
        nn = netifwrite(ether, chan, buf, n);
        if(nn >= 0)
            return nn;
        cb = parsecmd(buf, n);
        if(cb->f[0] && strcmp(cb->f[0], "nonblocking") == 0){
            if(cb->nf <= 1)
                onoff = 1;
            else
                onoff = atoi(cb->f[1]);
            qnoblock(ether->oq, onoff);
            free(cb);
            return n;
        }
        free(cb);
        if(ether->ctl != nil)
            return ether->ctl(ether, buf, n);

        error(Ebadctl);
```

```
    }

    if(n > ether->mtu)
        error(Etoobig);
    if(n < ether->minmtu)
        error(Etoosmall);

    bp = allocb(n);
    if(waserror()){
        freeb(bp);
        nexterror();
    }
    memmove(bp->rp, buf, n);
    memmove(bp->rp+Eaddrlen, ether->ea, Eaddrlen);
    poperror();
    bp->wp += n;

    return etheroq(ether, bp);
}
```

Uses `Eaddrlen` 27d, `NETTYPE` 240b, `Ndataqid` 240a, `etherxx-795` 28c, `netifwrite()` 246c, and `parsecmd()`.

⟨*function* `etherbwrite` 260a⟩≡ (264b)

```
static long
etherbwrite(Chan* chan, Block* bp, ulong)
{
    Ether *ether;
    long n;

    n = BLEN(bp);
    if(NETTYPE(chan->qid.path) != Ndataqid){
        if(waserror()) {
            freeb(bp);
            nexterror();
        }
        n = etherwrite(chan, bp->rp, n, 0);
        poperror();
        freeb(bp);
        return n;
    }
    ether = etherxx[chan->dev];

    if(n > ether->mtu){
        freeb(bp);
        error(Etoobig);
    }
    if(n < ether->minmtu){
        freeb(bp);
        error(Etoosmall);
    }

    return etheroq(ether, bp);
}
```

Uses `NETTYPE` 240b, `Ndataqid` 240a, and `etherxx-795` 28c.

⟨*global* `cards` 260b⟩≡ (264b)

```
static struct {
    char*   type;
    int (*reset)(Ether*);
} cards[MaxEther+1];
```

Uses `MaxEther` 28d and `__anon_struct_89` 260b.

⟨*function* addethercard 261a⟩≡ (264b)
```
void
addethercard(char* t, int (*r)(Ether*))
{
    static int ncard;

    if(ncard == MaxEther)
        panic("too many ether cards");
    cards[ncard].type = t;
    cards[ncard].reset = r;
    ncard++;
}
```
Uses MaxEther 28d and cards-796 260b.

⟨*function* parseether*(*(kernel/network/386/devether.c)*)* 261b⟩≡ (264b)
```
int
parseether(uchar *to, char *from)
{
    char nip[4];
    char *p;
    int i;

    p = from;
    for(i = 0; i < Eaddrlen; i++){
        if(*p == 0)
            return -1;
        nip[0] = *p++;
        if(*p == 0)
            return -1;
        nip[1] = *p++;
        nip[2] = 0;
        to[i] = strtoul(nip, 0, 16);
        if(*p == ':')
            p++;
    }
    return 0;
}
```
Uses Eaddrlen 27d.

⟨*function* etherprobe 261c⟩≡ (264b)
```
static Ether*
etherprobe(int cardno, int ctlrno)
{
    int i, lg;
    ulong mb, bsz;
    Ether *ether;
    char buf[128], name[32];

    ether = malloc(sizeof(Ether));
    if(ether == nil)
        error(Enomem);
    memset(ether, 0, sizeof(Ether));
    ether->ctlrno = ctlrno;
    ether->tbdf = BUSUNKNOWN;
    ether->mbps = 10;
    ether->minmtu = ETHERMINTU;
    ether->maxmtu = ETHERMAXTU;
    ether->mtu = ETHERMAXTU;

    if(cardno < 0){
```

```
	if(arch_isaconfig("ether", ctlrno, ether) == 0){
		free(ether);
		return nil;
	}
	for(cardno = 0; cards[cardno].type; cardno++){
		if(cistrcmp(cards[cardno].type, ether->type))
			continue;
		for(i = 0; i < ether->nopt; i++){
			if(strncmp(ether->opt[i], "ea=", 3))
				continue;
			if(parseether(ether->ea, &ether->opt[i][3]))
				memset(ether->ea, 0, Eaddrlen);
		}
		break;
	}
}

if(cardno >= MaxEther || cards[cardno].type == nil){
	free(ether);
	return nil;
}
if(cards[cardno].reset(ether) < 0){
	free(ether);
	return nil;
}

/*
 * IRQ2 doesn't really exist, it's used to gang the interrupt
 * controllers together. A device set to IRQ2 will appear on
 * the second interrupt controller as IRQ9.
 */
if(ether->irq == 2)
	ether->irq = 9;
snprint(name, sizeof(name), "ether%d", ctlrno);

/*
 * If ether->irq is <0, it is a hack to indicate no interrupt
 * used by ethersink.
 */
if(ether->irq >= 0)
	arch_intrenable(ether->irq, ether->interrupt, ether, ether->tbdf, name);

i = sprint(buf, "#l%d: %s: ", ctlrno, cards[cardno].type);
if(ether->mbps >= 1000)
	i += sprint(buf+i, "%dGbps", ether->mbps/1000);
else
	i += sprint(buf+i, "%dMbps", ether->mbps);
i += sprint(buf+i, " port 0x%luX irq %d", ether->port, ether->irq);
if(ether->mem)
	i += sprint(buf+i, " addr 0x%luX", ether->mem);
if(ether->size)
	i += sprint(buf+i, " size 0x%luX", ether->size);
i += sprint(buf+i, ": %2.2ux%2.2ux%2.2ux%2.2ux%2.2ux%2.2ux",
	ether->ea[0], ether->ea[1], ether->ea[2],
	ether->ea[3], ether->ea[4], ether->ea[5]);
sprint(buf+i, "\n");
print(buf);

/*
 * input queues are allocated by ../port/netif.c:/^openfile.
```

```
     * the size will be the last argument to netifinit() below.
     *
     * output queues should be small, to minimise ‘bufferbloat’,
     * which confuses tcp’s feedback loop.  at 1Gb/s, it only takes
     * ~15µs to transmit a full-sized non-jumbo packet.
     */

    /* compute log10(ether->mbps) into lg */
    for(lg = 0, mb = ether->mbps; mb >= 10; lg++)
        mb /= 10;
    if (lg > 14)            /* sanity cap; 2**(14+15) = 2^29 */
        lg = 14;

    /* allocate larger input queues for higher-speed interfaces */
    bsz = 1UL << (lg + 15);     /* 2^15 = 32K, bsz = 2^n × 32K */
    while (bsz > mainmem->maxsize / 8 && bsz > 128*1024)    /* sanity */
        bsz /= 2;
    netifinit(ether, name, Ntypes, bsz);

    if(ether->oq == nil)
        ether->oq = qopen(1 << (lg + 13), Qmsg, 0, 0);
    if(ether->oq == nil)
        panic("etherreset %s: can’t allocate output queue", name);

    ether->alen = Eaddrlen;
    memmove(ether->addr, ether->ea, Eaddrlen);
    memset(ether->bcast, 0xFF, Eaddrlen);

    return ether;
}
```

Uses `ETHERMAXTU` 240f, `ETHERMINTU` 240f, `Eaddrlen` 27d, `MaxEther` 28d, `Ntypes` 255a, `cards-796` 260b, and `netifinit()` 241b.

⟨*function* `etherreset` 263⟩≡ (264b)

```
static void
etherreset(void)
{
    Ether *ether;
    int cardno, ctlrno;

    for(ctlrno = 0; ctlrno < MaxEther; ctlrno++){
        if((ether = etherprobe(-1, ctlrno)) == nil)
            continue;
        etherxx[ctlrno] = ether;
    }

    if(getconf("*noetherprobe"))
        return;

    cardno = ctlrno = 0;
    while(cards[cardno].type != nil && ctlrno < MaxEther){
        if(etherxx[ctlrno] != nil){
            ctlrno++;
            continue;
        }
        if((ether = etherprobe(cardno, ctlrno)) == nil){
            cardno++;
            continue;
        }
        etherxx[ctlrno] = ether;
        ctlrno++;
```

```
    }
}
```
Uses MaxEther 28d, cards-796 260b, etherprobe() 261c, and etherxx-795 28c.

⟨*function* ethershutdown 264a⟩≡ (264b)
```
static void
ethershutdown(void)
{
    Ether *ether;
    int i;

    for(i = 0; i < MaxEther; i++){
        ether = etherxx[i];
        if(ether == nil)
            continue;
        if(ether->shutdown == nil) {
            print("#l%d: no shutdown function\n", i);
            continue;
        }
        (*ether->shutdown)(ether);
    }
}
```
Uses MaxEther 28d and etherxx-795 28c.

⟨kernel/network/386/devether.c 264b⟩≡
```
#include "u.h"
#include "../port/lib.h"
#include "../port/error.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"

#include "io.h"
#include "pool.h"
#include <ureg.h>
#include "../port/netif.h"
#include "../port/etherif.h"
```

⟨*global etherxx (kernel)* 28c⟩

⟨*function* etherattach 42c⟩

⟨*function* etherwalk 255e⟩

⟨*function* etherstat 255f⟩

⟨*function* etheropen 256a⟩

⟨*function* ethercreate 256b⟩

⟨*function* etherclose 256c⟩

⟨*function* etherread 256d⟩

⟨*function* etherbread 256e⟩

⟨*function* etherwstat 256f⟩

⟨*function* etherrtrace 257a⟩

⟨*function* `etheriq` 257b⟩

⟨*function* `etheroq` 258⟩

⟨*function* `etherwrite` 259⟩

⟨*function* `etherbwrite` 260a⟩

⟨*global* `cards` 260b⟩

⟨*function* `addethercard` 261a⟩

⟨*function* `parseether`*(*`(kernel/network/386/devether.c)`*)* 261b⟩

⟨*function* `etherprobe` 261c⟩

⟨*function* `etherreset` 263⟩

⟨*function* `ethershutdown` 264a⟩

⟨*global* `etherdevtab` 65f⟩

⟨*constant* `POLY` 265a⟩

⟨*function* `ethercrc` 265b⟩

⟨*constant* `POLY` 265a⟩≡ (264b)

```
#define POLY 0xedb88320
```

⟨*function* `ethercrc` 265b⟩≡ (264b)

```
/* really slow 32 bit crc for ethers */
ulong
ethercrc(uchar *p, int len)
{
  int i, j;
  ulong crc, b;

  crc = 0xffffffff;
  for(i = 0; i < len; i++){
      b = *p++;
      for(j = 0; j < 8; j++){
          crc = (crc>>1) ^ (((crc^b) & 1) ? POLY : 0);
          b >>= 1;
      }
  }
  return crc;
}
```

Uses `POLY-797` 265a.

I.6.3 `kernel/network/386/ether8390.h`

⟨*constant* `Dp8390BufSz` 265c⟩≡ (266a)

```
#define Dp8390BufSz 256
```

⟨*function* `rdread` 265d⟩≡ (266a)

```
static void
rdread(Dp8390* ctlr, void* to, int len)
{
    switch(ctlr->width){
```

```
    default:
        panic("dp8390 rdread: width %d\n", ctlr->width);
        break;

    case 2:
        inss(ctlr->data, to, len/2);
        break;

    case 1:
        insb(ctlr->data, to, len);
        break;
    }
}
```

⟨kernel/network/386/ether8390.h 266a⟩≡

```
typedef struct Dp8390 Dp8390;
⟨struct Dp8390 196⟩

⟨constant Dp8390BufSz 265c⟩

extern int dp8390reset(Ether*);
extern void *dp8390read(Dp8390*, void*, ulong, ulong);
extern void dp8390getea(Ether*, uchar*);
extern void dp8390setea(Ether*);

⟨macro regr 202b⟩
⟨macro regw 202a⟩

⟨function rdread 265d⟩

⟨function rdwrite 205a⟩
```

Uses Dp8390 196.

I.6.4 kernel/network/386/ether8390.c

⟨*enum* _anon_ (kernel/network/386/ether8390.c)0 266b⟩≡ (269)

```
enum {      /* NIC core registers */
    Cr  = 0x00,  /* command register, all pages */

                 /* Page 0, read */
    Clda0  = 0x01,  /* current local DMA address 0 */
    Clda1  = 0x02,  /* current local DMA address 1 */
    Bnry  = 0x03,  /* boundary pointer (R/W) */
    Tsr  = 0x04,  /* transmit status register */
    Ncr  = 0x05,  /* number of collisions register */
    Fifo  = 0x06,  /* FIFO */
    Isr  = 0x07,  /* interrupt status register (R/W) */
    Crda0  = 0x08,  /* current remote DMA address 0 */
    Crda1  = 0x09,  /* current remote DMA address 1 */
    Rsr  = 0x0C,  /* receive status register */
    Ref0  = 0x0D,  /* frame alignment errors */
    Ref1  = 0x0E,  /* CRC errors */
    Ref2  = 0x0F,  /* missed packet errors */

                 /* Page 0, write */
    Pstart  = 0x01,  /* page start register */
    Pstop  = 0x02,  /* page stop register */
    Tpsr  = 0x04,  /* transmit page start address */
```

```
    Tbcr0  = 0x05,  /* transmit byte count register 0 */
    Tbcr1  = 0x06,  /* transmit byte count register 1 */
    Rsar0  = 0x08,  /* remote start address register 0 */
    Rsar1  = 0x09,  /* remote start address register 1 */
    Rbcr0  = 0x0A,  /* remote byte count register 0 */
    Rbcr1  = 0x0B,  /* remote byte count register 1 */
    Rcr  = 0x0C,  /* receive configuration register */
    Tcr  = 0x0D,  /* transmit configuration register */
    Dcr  = 0x0E,  /* data configuration register */
    Imr  = 0x0F,  /* interrupt mask */

                    /* Page 1, read/write */
    Par0  = 0x01,  /* physical address register 0 */
    Curr  = 0x07,  /* current page register */
    Mar0  = 0x08,  /* multicast address register 0 */
};
```

⟨*enum* _anon_ (kernel/network/386/ether8390.c)1 267a⟩≡ (269)

```
enum {     /* Cr */
    Stp  = 0x01,  /* stop */
    Sta  = 0x02,  /* start */
    Txp  = 0x04,  /* transmit packet */
    Rd0  = 0x08,  /* remote DMA command */
    Rd1  = 0x10,
    Rd2  = 0x20,
    RdREAD  = Rd0,  /* remote read */
    RdWRITE  = Rd1,  /* remote write */
    RdSEND  = Rd1|Rd0, /* send packet */
    RdABORT  = Rd2,  /* abort/complete remote DMA */
    Ps0  = 0x40,  /* page select */
    Ps1  = 0x80,
    Page0  = 0x00,
    Page1  = Ps0,
    Page2  = Ps1,
};
```

Uses Ps0-421 267a, Ps1-422 267a, Rd0-414 267a, Rd1-415 267a, and Rd2-416 267a.

⟨*enum* _anon_ (kernel/network/386/ether8390.c)2 267b⟩≡ (269)

```
enum {     /* Isr/Imr */
    Prx  = 0x01,  /* packet received */
    Ptx  = 0x02,  /* packet transmitted */
    Rxe  = 0x04,  /* receive error */
    Txe  = 0x08,  /* transmit error */
    Ovw  = 0x10,  /* overwrite warning */
    Cnt  = 0x20,  /* counter overflow */
    Rdc  = 0x40,  /* remote DMA complete */
    Rst  = 0x80,  /* reset status */
};
```

⟨*enum* _anon_ (kernel/network/386/ether8390.c)3 267c⟩≡ (269)

```
enum {     /* Dcr */
    Wts  = 0x01,  /* word transfer select */
    Bos  = 0x02,  /* byte order select */
    Las  = 0x04,  /* long address select */
    Ls  = 0x08,  /* loopback select */
    Arm  = 0x10,  /* auto-initialise remote */
    Ft0  = 0x20,  /* FIFO threshold select */
    Ft1  = 0x40,
    Ft1WORD  = 0x00,
    Ft2WORD  = Ft0,
```

```
    Ft4WORD  = Ft1,
    Ft6WORD  = Ft1|Ft0,
};
```
Uses Ft0-439 267c and Ft1-440 267c.

⟨*enum* _anon_ (kernel/network/386/ether8390.c)4 268a⟩≡ (269)
```
enum {     /* Tcr */
    Crc  = 0x01,  /* inhibit CRC */
    Lb0  = 0x02,  /* encoded loopback control */
    Lb1  = 0x04,
    LpbkNORMAL = 0x00,  /* normal operation */
    LpbkNIC  = Lb0,  /* internal NIC module loopback */
    LpbkENDEC = Lb1,  /* internal ENDEC module loopback */
    LpbkEXTERNAL = Lb1|Lb0, /* external loopback */
    Atd  = 0x08,  /* auto transmit disable */
    Ofst  = 0x10,  /* collision offset enable */
};
```
Uses Lb0-446 268a and Lb1-447 268a.

⟨*enum* _anon_ (kernel/network/386/ether8390.c)5 268b⟩≡ (269)
```
enum {     /* Tsr */
    Ptxok  = 0x01,  /* packet transmitted */
    Col  = 0x04,  /* transmit collided */
    Abt  = 0x08,  /* tranmit aborted */
    Crs  = 0x10,  /* carrier sense lost */
    Fu  = 0x20,  /* FIFO underrun */
    Cdh  = 0x40,  /* CD heartbeat */
    Owc  = 0x80,  /* out of window collision */
};
```

⟨*enum* _anon_ (kernel/network/386/ether8390.c)6 268c⟩≡ (269)
```
enum {     /* Rcr */
    Sep  = 0x01,  /* save errored packets */
    Ar  = 0x02,  /* accept runt packets */
    Ab  = 0x04,  /* accept broadcast */
    Am  = 0x08,  /* accept multicast */
    Pro  = 0x10,  /* promiscuous physical */
    Mon  = 0x20,  /* monitor mode */
};
```

⟨*enum* _anon_ (kernel/network/386/ether8390.c)7 268d⟩≡ (269)
```
enum {     /* Rsr */
    Prxok  = 0x01,  /* packet received intact */
    Crce  = 0x02,  /* CRC error */
    Fae  = 0x04,  /* frame alignment error */
    Fo  = 0x08,  /* FIFO overrun */
    Mpa  = 0x10,  /* missed packet */
    Phy  = 0x20,  /* physical/multicast address */
    Dis  = 0x40,  /* receiver disabled */
    Dfr  = 0x80,  /* deferring */
};
```

⟨*struct* Hdr *(*(kernel/network/386/ether8390.c)*)* 268e⟩≡ (269)
```
struct Hdr {
    uchar status;
    uchar next;
    uchar len0;
    uchar len1;
};
```

```
⟨kernel/network/386/ether8390.c 269⟩≡
  /*
   * National Semiconductor DP8390 and clone
   * Network Interface Controller.
   */
  #include "u.h"
  #include "../port/lib.h"
  #include "mem.h"
  #include "dat.h"
  #include "fns.h"
  #include "io.h"
  #include "../port/error.h"
  #include "../port/netif.h"
  #include "../port/etherif.h"

  #include "ether8390.h"

  ⟨enum _anon_ (kernel/network/386/ether8390.c)0 266b⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)1 267a⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)2 267b⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)3 267c⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)4 268a⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)5 268b⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)6 268c⟩

  ⟨enum _anon_ (kernel/network/386/ether8390.c)7 268d⟩

  typedef struct Hdr Hdr;


  ⟨struct Hdr((kernel/network/386/ether8390.c)) 268e⟩


  ⟨function dp8390setea 200b⟩

  ⟨function _dp8390read 208b⟩

  ⟨function dp8390read 199a⟩

  ⟨function dp8390write 204⟩

  ⟨function ringinit 200a⟩

  ⟨function getcurr 208a⟩

  ⟨function receive((kernel/network/386/ether8390.c)) 207⟩

  ⟨function txstart 203⟩

  ⟨function transmit((kernel/network/386/ether8390.c)) 202c⟩

  ⟨function overflow 209⟩

  ⟨function interrupt((kernel/network/386/ether8390.c)) 205b⟩
```

⟨*global* allmar 211a⟩

⟨*function* setfilter 211c⟩

⟨*function* promiscuous 212a⟩

⟨*function* setbit 211b⟩

⟨*global* reverse 210b⟩

⟨*function* multicast 210c⟩

⟨*function* attach 201c⟩

⟨*function* disable 201a⟩

⟨*function* shutdown 201b⟩

⟨*function* dp8390reset 199b⟩

Uses Hdr 268e.

I.6.5 kernel/network/386/ether2000.c

⟨*enum* _anon_ (kernel/network/386/ether2000.c) 270a⟩≡ (270b)

```
enum {
    Data  = 0x10,  /* offset from I/O base of data port */
    Reset  = 0x1F,  /* offset from I/O base of reset port */
};
```

⟨kernel/network/386/ether2000.c 270b⟩≡

```
#include "u.h"
#include "../port/lib.h"
#include "../port/error.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"

#include "io.h"
#include "../port/netif.h"
#include "../port/etherif.h"

#include "ether8390.h"

/*
 * Driver written for the 'Notebook Computer Ethernet LAN Adapter',
 * a plug-in to the bus-slot on the rear of the Gateway NOMAD 425DXL
 * laptop. The manual says NE2000 compatible.
 * The interface appears to be pretty well described in the National
 * Semiconductor Local Area Network Databook (1992) as one of the
 * AT evaluation cards.
 *
 * The NE2000 is really just a DP8390[12] plus a data port
 * and a reset port.
 */
```

⟨*enum* _anon_ (kernel/network/386/ether2000.c) 270a⟩

```
typedef struct Ctlr Ctlr;
```

⟨*struct* Ctlr *(*(kernel/network/386/ether2000.c)*)* 212b⟩

⟨*global* ctlrhead *(*(kernel/network/386/ether2000.c)*)* 212c⟩
⟨*global* ctlrtail *(*(kernel/network/386/ether2000.c)*)* 212d⟩

⟨*global* ne2000pci 213a⟩

⟨*function* ne2000match 213b⟩

⟨*function* ne2000pnp 212e⟩

⟨*function* ne2000reset 197b⟩

⟨*function* ether2000link 197a⟩

Uses Ctlr 212b.

I.6.6 kernel/network/386/etherigbe.c

⟨*enum* _anon_ (kernel/network/386/etherigbe.c) 271a⟩≡ (307b)

```
enum {
    i82542      = (0x1000<<16)|0x8086,
    i82543gc    = (0x1004<<16)|0x8086,
    i82544ei    = (0x1008<<16)|0x8086,
    i82544eif   = (0x1009<<16)|0x8086,
    i82544gc    = (0x100d<<16)|0x8086,
    i82540em    = (0x100E<<16)|0x8086,
    i82540eplp  = (0x101E<<16)|0x8086,
    i82545em    = (0x100F<<16)|0x8086,
    i82545gmc   = (0x1026<<16)|0x8086,
    i82547ei    = (0x1019<<16)|0x8086,
    i82547gi    = (0x1075<<16)|0x8086,
    i82541ei    = (0x1013<<16)|0x8086,
    i82541gi    = (0x1076<<16)|0x8086,
    i82541gi2   = (0x1077<<16)|0x8086,
    i82541pi    = (0x107c<<16)|0x8086,
    i82546gb    = (0x1079<<16)|0x8086,
    i82546eb    = (0x1010<<16)|0x8086,
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)2 271b⟩≡ (307b)

```
enum {
    Ctrl        = 0x00000000,   /* Device Control */
    Ctrldup     = 0x00000004,   /* Device Control Duplicate */
    Status      = 0x00000008,   /* Device Status */
    Eecd        = 0x00000010,   /* EEPROM/Flash Control/Data */
    Ctrlext     = 0x00000018,   /* Extended Device Control */
    Mdic        = 0x00000020,   /* MDI Control */
    Fcal        = 0x00000028,   /* Flow Control Address Low */
    Fcah        = 0x0000002C,   /* Flow Control Address High */
    Fct     = 0x00000030,   /* Flow Control Type */
    Icr     = 0x000000C0,   /* Interrupt Cause Read */
    Ics     = 0x000000C8,   /* Interrupt Cause Set */
    Ims     = 0x000000D0,   /* Interrupt Mask Set/Read */
    Imc     = 0x000000D8,   /* Interrupt mask Clear */
    Rctl        = 0x00000100,   /* Receive Control */
    Fcttv       = 0x00000170,   /* Flow Control Transmit Timer Value */
    Txcw        = 0x00000178,   /* Transmit Configuration Word */
    Rxcw        = 0x00000180,   /* Receive Configuration Word */
    /* on the oldest cards (8254[23]), the Mta register is at 0x200 */
```

```
    Tctl        = 0x00000400,   /* Transmit Control */
    Tipg        = 0x00000410,   /* Transmit IPG */
    Tbt     = 0x00000448,   /* Transmit Burst Timer */
    Ait     = 0x00000458,   /* Adaptive IFS Throttle */
    Fcrtl       = 0x00002160,   /* Flow Control RX Threshold Low */
    Fcrth       = 0x00002168,   /* Flow Control Rx Threshold High */
    Rdfh        = 0x00002410,   /* Receive data fifo head */
    Rdft        = 0x00002418,   /* Receive data fifo tail */
    Rdfhs       = 0x00002420,   /* Receive data fifo head saved */
    Rdfts       = 0x00002428,   /* Receive data fifo tail saved */
    Rdfpc       = 0x00002430,   /* Receive data fifo packet count */
    Rdbal       = 0x00002800,   /* Rd Base Address Low */
    Rdbah       = 0x00002804,   /* Rd Base Address High */
    Rdlen       = 0x00002808,   /* Receive Descriptor Length */
    Rdh     = 0x00002810,   /* Receive Descriptor Head */
    Rdt     = 0x00002818,   /* Receive Descriptor Tail */
    Rdtr        = 0x00002820,   /* Receive Descriptor Timer Ring */
    Rxdctl      = 0x00002828,   /* Receive Descriptor Control */
    Radv        = 0x0000282C,   /* Receive Interrupt Absolute Delay Timer */
    Txdmac      = 0x00003000,   /* Transfer DMA Control */
    Ett     = 0x00003008,   /* Early Transmit Control */
    Tdfh        = 0x00003410,   /* Transmit data fifo head */
    Tdft        = 0x00003418,   /* Transmit data fifo tail */
    Tdfhs       = 0x00003420,   /* Transmit data Fifo Head saved */
    Tdfts       = 0x00003428,   /* Transmit data fifo tail saved */
    Tdfpc       = 0x00003430,   /* Trasnmit data Fifo packet count */
    Tdbal       = 0x00003800,   /* Td Base Address Low */
    Tdbah       = 0x00003804,   /* Td Base Address High */
    Tdlen       = 0x00003808,   /* Transmit Descriptor Length */
    Tdh     = 0x00003810,   /* Transmit Descriptor Head */
    Tdt     = 0x00003818,   /* Transmit Descriptor Tail */
    Tidv        = 0x00003820,   /* Transmit Interrupt Delay Value */
    Txdctl      = 0x00003828,   /* Transmit Descriptor Control */
    Tadv        = 0x0000382C,   /* Transmit Interrupt Absolute Delay Timer */

    Statistics  = 0x00004000,   /* Start of Statistics Area */
    Gorcl       = 0x88/4,   /* Good Octets Received Count */
    Gotcl       = 0x90/4,   /* Good Octets Transmitted Count */
    Torl        = 0xC0/4,   /* Total Octets Received */
    Totl        = 0xC8/4,   /* Total Octets Transmitted */
    Nstatistics = 64,

    Rxcsum      = 0x00005000,   /* Receive Checksum Control */
    Mta     = 0x00005200,   /* Multicast Table Array */
    Ral     = 0x00005400,   /* Receive Address Low */
    Rah     = 0x00005404,   /* Receive Address High */
    Manc        = 0x00005820,   /* Management Control */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)3 272⟩≡ (307b)

```
enum {                   /* Ctrl */
    Bem     = 0x00000002,   /* Big Endian Mode */
    Prior       = 0x00000004,   /* Priority on the PCI bus */
    Lrst        = 0x00000008,   /* Link Reset */
    Asde        = 0x00000020,   /* Auto-Speed Detection Enable */
    Slu     = 0x00000040,   /* Set Link Up */
    Ilos        = 0x00000080,   /* Invert Loss of Signal (LOS) */
    SspeedMASK  = 0x00000300,   /* Speed Selection */
    SspeedSHIFT = 8,
    Sspeed10    = 0x00000000,   /* 10Mb/s */
```

```
    Sspeed100    = 0x00000100,   /* 100Mb/s */
    Sspeed1000   = 0x00000200,   /* 1000Mb/s */
    Frcspd       = 0x00000800,   /* Force Speed */
    Frcdplx      = 0x00001000,   /* Force Duplex */
    SwdpinsloMASK    = 0x003C0000,   /* Software Defined Pins - lo nibble */
    SwdpinsloSHIFT   = 18,
    SwdpioloMASK     = 0x03C00000,   /* Software Defined Pins - I or O */
    SwdpioloSHIFT    = 22,
    Devrst       = 0x04000000,   /* Device Reset */
    Rfce         = 0x08000000,   /* Receive Flow Control Enable */
    Tfce         = 0x10000000,   /* Transmit Flow Control Enable */
    Vme      = 0x40000000,   /* VLAN Mode Enable */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)4 273a⟩≡ (307b)

```
/*
 * can't find Tckok nor Rbcok in any Intel docs,
 * but even 82543gc docs define Lanid.
 */
enum {                  /* Status */
    Lu       = 0x00000002,   /* Link Up */
    Lanid        = 0x0000000C,   /* mask for Lan ID. (function id) */
//  Tckok        = 0x00000004,   /* Transmit clock is running */
//  Rbcok        = 0x00000008,   /* Receive clock is running */
    Txoff        = 0x00000010,   /* Transmission Paused */
    Tbimode      = 0x00000020,   /* TBI Mode Indication */
    LspeedMASK   = 0x000000C0,   /* Link Speed Setting */
    LspeedSHIFT = 6,
    Lspeed10     = 0x00000000,   /* 10Mb/s */
    Lspeed100    = 0x00000040,   /* 100Mb/s */
    Lspeed1000   = 0x00000080,   /* 1000Mb/s */
    Mtxckok      = 0x00000400,   /* MTX clock is running */
    Pci66        = 0x00000800,   /* PCI Bus speed indication */
    Bus64        = 0x00001000,   /* PCI Bus width indication */
    Pcixmode     = 0x00002000,   /* PCI-X mode */
    PcixspeedMASK    = 0x0000C000,   /* PCI-X bus speed */
    PcixspeedSHIFT   = 14,
    Pcix66       = 0x00000000,   /* 50-66MHz */
    Pcix100      = 0x00004000,   /* 66-100MHz */
    Pcix133      = 0x00008000,   /* 100-133MHz */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)5 273b⟩≡ (307b)

```
enum {                  /* Ctrl and Status */
    Fd       = 0x00000001,   /* Full-Duplex */
    AsdvMASK     = 0x00000300,
    AsdvSHIFT    = 8,
    Asdv10       = 0x00000000,   /* 10Mb/s */
    Asdv100      = 0x00000100,   /* 100Mb/s */
    Asdv1000     = 0x00000200,   /* 1000Mb/s */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)6 273c⟩≡ (307b)

```
enum {                  /* Eecd */
    Sk       = 0x00000001,   /* Clock input to the EEPROM */
    Cs       = 0x00000002,   /* Chip Select */
    Di       = 0x00000004,   /* Data Input to the EEPROM */
    Do       = 0x00000008,   /* Data Output from the EEPROM */
    Areq         = 0x00000040,   /* EEPROM Access Request */
    Agnt         = 0x00000080,   /* EEPROM Access Grant */
```

```
    Eepresent   = 0x00000100,   /* EEPROM Present */
    Eesz256     = 0x00000200,   /* EEPROM is 256 words not 64 */
    Eeszaddr    = 0x00000400,   /* EEPROM size for 8254[17] */
    Spi     = 0x00002000,   /* EEPROM is SPI not Microwire */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)7 274a⟩≡ (307b)

```
enum {                  /* Ctrlext */
    Gpien       = 0x0000000F,   /* General Purpose Interrupt Enables */
    SwdpinshiMASK   = 0x000000F0,   /* Software Defined Pins - hi nibble */
    SwdpinshiSHIFT  = 4,
    SwdpiohiMASK    = 0x00000F00,   /* Software Defined Pins - I or O */
    SwdpiohiSHIFT   = 8,
    Asdchk      = 0x00001000,   /* ASD Check */
    Eerst       = 0x00002000,   /* EEPROM Reset */
    Ips     = 0x00004000,   /* Invert Power State */
    Spdbyps     = 0x00008000,   /* Speed Select Bypass */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)8 274b⟩≡ (307b)

```
enum {                  /* EEPROM content offsets */
    Ea      = 0x00,     /* Ethernet Address */
    Cf      = 0x03,     /* Compatibility Field */
    Pba     = 0x08,     /* Printed Board Assembly number */
    Icw1        = 0x0A,     /* Initialization Control Word 1 */
    Sid     = 0x0B,     /* Subsystem ID */
    Svid        = 0x0C,     /* Subsystem Vendor ID */
    Did     = 0x0D,     /* Device ID */
    Vid     = 0x0E,     /* Vendor ID */
    Icw2        = 0x0F,     /* Initialization Control Word 2 */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)9 274c⟩≡ (307b)

```
enum {                  /* Mdic */
    MDIdMASK    = 0x0000FFFF,   /* Data */
    MDIdSHIFT   = 0,
    MDIrMASK    = 0x001F0000,   /* PHY Register Address */
    MDIrSHIFT   = 16,
    MDIpMASK    = 0x03E00000,   /* PHY Address */
    MDIpSHIFT   = 21,
    MDIwop      = 0x04000000,   /* Write Operation */
    MDIrop      = 0x08000000,   /* Read Operation */
    MDIready    = 0x10000000,   /* End of Transaction */
    MDIie       = 0x20000000,   /* Interrupt Enable */
    MDIe        = 0x40000000,   /* Error */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)10 274d⟩≡ (307b)

```
enum {                  /* Icr, Ics, Ims, Imc */
    Txdw        = 0x00000001,   /* Transmit Descriptor Written Back */
    Txqe        = 0x00000002,   /* Transmit Queue Empty */
    Lsc     = 0x00000004,   /* Link Status Change */
    Rxseq       = 0x00000008,   /* Receive Sequence Error */
    Rxdmt0      = 0x00000010,   /* Rd Minimum Threshold Reached */
    Rxo     = 0x00000040,   /* Receiver Overrun */
    Rxt0        = 0x00000080,   /* Receiver Timer Interrupt */
    Mdac        = 0x00000200,   /* MDIO Access Completed */
    Rxcfg       = 0x00000400,   /* Receiving /C/ ordered sets */
    Gpi0        = 0x00000800,   /* General Purpose Interrupts */
    Gpi1        = 0x00001000,
```

```
    Gpi2        = 0x00002000,
    Gpi3        = 0x00004000,
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)11 275a⟩≡ (307b)

```
/*
 * The Mdic register isn't implemented on the 82543GC,
 * the software defined pins are used instead.
 * These definitions work for the Intel PRO/1000 T Server Adapter.
 * The direction pin bits are read from the EEPROM.
 */
enum {
    Mdd     = ((1<<2)<<SwdpinsloSHIFT), /* data */
    Mddo        = ((1<<2)<<SwdpioloSHIFT),  /* pin direction */
    Mdc     = ((1<<3)<<SwdpinsloSHIFT), /* clock */
    Mdco        = ((1<<3)<<SwdpioloSHIFT),  /* pin direction */
    Mdr     = ((1<<0)<<SwdpinshiSHIFT), /* reset */
    Mdro        = ((1<<0)<<SwdpiohiSHIFT),  /* pin direction */
};
```

Uses SwdpinshiSHIFT-613 274a, SwdpinsloSHIFT-570 272, SwdpiohiSHIFT-615 274a, and SwdpioloSHIFT-572 272.

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)12 275b⟩≡ (307b)

```
enum {                  /* Txcw */
    TxcwFd      = 0x00000020,   /* Full Duplex */
    TxcwHd      = 0x00000040,   /* Half Duplex */
    TxcwPauseMASK   = 0x00000180,   /* Pause */
    TxcwPauseSHIFT  = 7,
    TxcwPs      = (1<<TxcwPauseSHIFT),  /* Pause Supported */
    TxcwAs      = (2<<TxcwPauseSHIFT),  /* Asymmetric FC desired */
    TxcwRfiMASK = 0x00003000,   /* Remote Fault Indication */
    TxcwRfiSHIFT    = 12,
    TxcwNpr     = 0x00008000,   /* Next Page Request */
    TxcwConfig  = 0x40000000,   /* Transmit COnfig Control */
    TxcwAne     = 0x80000000,   /* Auto-Negotiation Enable */
};
```

Uses TxcwPauseSHIFT-662 275b.

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)13 275c⟩≡ (307b)

```
enum {                  /* Rxcw */
    Rxword      = 0x0000FFFF,   /* Data from auto-negotiation process */
    Rxnocarrier = 0x04000000,   /* Carrier Sense indication */
    Rxinvalid   = 0x08000000,   /* Invalid Symbol during configuration */
    Rxchange    = 0x10000000,   /* Change to the Rxword indication */
    Rxconfig    = 0x20000000,   /* /C/ order set reception indication */
    Rxsync      = 0x40000000,   /* Lost bit synchronization indication */
    Anc     = 0x80000000,   /* Auto Negotiation Complete */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)14 275d⟩≡ (307b)

```
enum {                  /* Rctl */
    Rrst        = 0x00000001,   /* Receiver Software Reset */
    Ren     = 0x00000002,   /* Receiver Enable */
    Sbp     = 0x00000004,   /* Store Bad Packets */
    Upe     = 0x00000008,   /* Unicast Promiscuous Enable */
    Mpe     = 0x00000010,   /* Multicast Promiscuous Enable */
    Lpe     = 0x00000020,   /* Long Packet Reception Enable */
    LbmMASK     = 0x000000C0,   /* Loopback Mode */
    LbmOFF      = 0x00000000,   /* No Loopback */
    LbmTBI      = 0x00000040,   /* TBI Loopback */
    LbmMII      = 0x00000080,   /* GMII/MII Loopback */
```

```
    LbmXCVR     = 0x000000C0,   /* Transceiver Loopback */
    RdtmsMASK   = 0x00000300,   /* Rd Minimum Threshold Size */
    RdtmsHALF   = 0x00000000,   /* Threshold is 1/2 Rdlen */
    RdtmsQUARTER    = 0x00000100,   /* Threshold is 1/4 Rdlen */
    RdtmsEIGHTH = 0x00000200,   /* Threshold is 1/8 Rdlen */
    MoMASK      = 0x00003000,   /* Multicast Offset */
    Mo47b36     = 0x00000000,   /* bits [47:36] of received address */
    Mo46b35     = 0x00001000,   /* bits [46:35] of received address */
    Mo45b34     = 0x00002000,   /* bits [45:34] of received address */
    Mo43b32     = 0x00003000,   /* bits [43:32] of received address */
    Bam     = 0x00008000,   /* Broadcast Accept Mode */
    BsizeMASK   = 0x00030000,   /* Receive Buffer Size */
    Bsize2048   = 0x00000000,   /* Bsex = 0 */
    Bsize1024   = 0x00010000,   /* Bsex = 0 */
    Bsize512    = 0x00020000,   /* Bsex = 0 */
    Bsize256    = 0x00030000,   /* Bsex = 0 */
    Bsize16384  = 0x00010000,   /* Bsex = 1 */
    Vfe     = 0x00040000,   /* VLAN Filter Enable */
    Cfien       = 0x00080000,   /* Canonical Form Indicator Enable */
    Cfi     = 0x00100000,   /* Canonical Form Indicator value */
    Dpf     = 0x00400000,   /* Discard Pause Frames */
    Pmcf        = 0x00800000,   /* Pass MAC Control Frames */
    Bsex        = 0x02000000,   /* Buffer Size Extension */
    Secrc       = 0x04000000,   /* Strip CRC from incoming packet */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)15 276a⟩≡ (307b)

```
enum {                  /* Tctl */
    Trst        = 0x00000001,   /* Transmitter Software Reset */
    Ten     = 0x00000002,   /* Transmit Enable */
    Psp     = 0x00000008,   /* Pad Short Packets */
    CtMASK      = 0x00000FF0,   /* Collision Threshold */
    CtSHIFT     = 4,
    ColdMASK    = 0x003FF000,   /* Collision Distance */
    ColdSHIFT   = 12,
    Swxoff      = 0x00400000,   /* Sofware XOFF Transmission */
    Pbe     = 0x00800000,   /* Packet Burst Enable */
    Rtlc        = 0x01000000,   /* Re-transmit on Late Collision */
    Nrtu        = 0x02000000,   /* No Re-transmit on Underrrun */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)16 276b⟩≡ (307b)

```
enum {                  /* [RT]xdctl */
    PthreshMASK = 0x0000003F,   /* Prefetch Threshold */
    PthreshSHIFT    = 0,
    HthreshMASK = 0x00003F00,   /* Host Threshold */
    HthreshSHIFT    = 8,
    WthreshMASK = 0x003F0000,   /* Writeback Threshold */
    WthreshSHIFT    = 16,
    Gran        = 0x01000000,   /* Granularity */
    LthreshMASK = 0xFE000000,   /* Low Threshold */
    LthreshSHIFT    = 25,
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)17 276c⟩≡ (307b)

```
enum {                  /* Rxcsum */
    PcssMASK    = 0x000000FF,   /* Packet Checksum Start */
    PcssSHIFT   = 0,
    Ipofl       = 0x00000100,   /* IP Checksum Off-load Enable */
    Tuofl       = 0x00000200,   /* TCP/UDP Checksum Off-load Enable */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)18 277a⟩≡ (307b)
```
enum {                  /* Manc */
    Arpen       = 0x00002000,   /* Enable ARP Request Filtering */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)19 277b⟩≡ (307b)
```
enum {                  /* Receive Delay Timer Ring */
    DelayMASK   = 0x0000FFFF,   /* delay timer in 1.024nS increments */
    DelaySHIFT  = 0,
    Fpd     = 0x80000000,   /* Flush partial Descriptor Block */
};
```

⟨*struct* Rd 277c⟩≡ (307b)
```
typedef struct Rd {         /* Receive Descriptor */
    uint    addr[2];
    ushort  length;
    ushort  checksum;
    uchar   status;
    uchar   errors;
    ushort  special;
} Rd;
```
Uses Rd 277c.

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)20 277d⟩≡ (307b)
```
enum {                  /* Rd status */
    Rdd     = 0x01,     /* Descriptor Done */
    Reop        = 0x02,     /* End of Packet */
    Ixsm        = 0x04,     /* Ignore Checksum Indication */
    Vp      = 0x08,     /* Packet is 802.1Q (matched VET) */
    Tcpcs       = 0x20,     /* TCP Checksum Calculated on Packet */
    Ipcs        = 0x40,     /* IP Checksum Calculated on Packet */
    Pif     = 0x80,     /* Passed in-exact filter */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)21 277e⟩≡ (307b)
```
enum {                  /* Rd errors */
    Ce      = 0x01,     /* CRC Error or Alignment Error */
    Se      = 0x02,     /* Symbol Error */
    Seq     = 0x04,     /* Sequence Error */
    Cxe     = 0x10,     /* Carrier Extension Error */
    Tcpe        = 0x20,     /* TCP/UDP Checksum Error */
    Ipe     = 0x40,     /* IP Checksum Error */
    Rxe     = 0x80,     /* RX Data Error */
};
```

⟨*struct* Td 277f⟩≡ (307b)
```
struct Td {             /* Transmit Descriptor */
    union {
        uint    addr[2];    /* Data */
        struct {        /* Context */
            uchar   ipcss;
            uchar   ipcso;
            ushort  ipcse;
            uchar   tucss;
            uchar   tucso;
            ushort  tucse;
        };
    };
    uint    control;
    uint    status;
};
```
Uses __anon_struct_83 277f and __anon_struct_84 277f.

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)22 278a⟩≡ (307b)

```
enum {                  /* Td control */
    LenMASK     = 0x000FFFFF,   /* Data/Packet Length Field */
    LenSHIFT    = 0,
    DtypeCD     = 0x00000000,   /* Data Type 'Context Descriptor' */
    DtypeDD     = 0x00100000,   /* Data Type 'Data Descriptor' */
    PtypeTCP    = 0x01000000,   /* TCP/UDP Packet Type (CD) */
    Teop        = 0x01000000,   /* End of Packet (DD) */
    PtypeIP     = 0x02000000,   /* IP Packet Type (CD) */
    Ifcs        = 0x02000000,   /* Insert FCS (DD) */
    Tse     = 0x04000000,   /* TCP Segmentation Enable */
    Rs      = 0x08000000,   /* Report Status */
    Rps     = 0x10000000,   /* Report Status Sent */
    Dext        = 0x20000000,   /* Descriptor Extension */
    Vle     = 0x40000000,   /* VLAN Packet Enable */
    Ide     = 0x80000000,   /* Interrupt Delay Enable */
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)23 278b⟩≡ (307b)

```
enum {                  /* Td status */
    Tdd     = 0x00000001,   /* Descriptor Done */
    Ec      = 0x00000002,   /* Excess Collisions */
    Lc      = 0x00000004,   /* Late Collision */
    Tu      = 0x00000008,   /* Transmit Underrun */
    Iixsm       = 0x00000100,   /* Insert IP Checksum */
    Itxsm       = 0x00000200,   /* Insert TCP/UDP Checksum */
    HdrlenMASK  = 0x0000FF00,   /* Header Length (Tse) */
    HdrlenSHIFT = 8,
    VlanMASK    = 0x0FFF0000,   /* VLAN Identifier */
    VlanSHIFT   = 16,
    Tcfi        = 0x10000000,   /* Canonical Form Indicator */
    PriMASK     = 0xE0000000,   /* User Priority */
    PriSHIFT    = 29,
    MssMASK     = 0xFFFF0000,   /* Maximum Segment Size (Tse) */
    MssSHIFT    = 16,
};
```

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)24 278c⟩≡ (307b)

```
enum {
    Nrd     = 256,      /* multiple of 8 */
    Ntd     = 64,       /* multiple of 8 */
    Nrb     = 1024,     /* private receive buffers per Ctlr */
    Rbsz        = 2048,
};
```

⟨*struct* CtlrEtherIgbe 278d⟩≡ (307b)

```
struct CtlrEtherIgbe {
    int port;
    Pcidev* pcidev;
    CtlrEtherIgbe*  next;
    Ether*  edev;
    int active;
    int started;
    int id;
    int cls;
    ushort  eeprom[0x40];

    QLock   alock;          /* attach */
    void*   alloc;          /* receive/transmit descriptors */
    int nrd;
```

```
    int ntd;
    int nrb;            /* # bufs this Ctlr has in the pool */

    int*    nic;
    Lock    imlock;
    int im;         /* interrupt mask */

    Mii*    mii;
    Rendez  lrendez;
    int lim;

    int link;

    QLock   slock;
    uint    statistics[Nstatistics];
    uint    lsleep;
    uint    lintr;
    uint    rsleep;
    uint    rintr;
    uint    txdw;
    uint    tintr;
    uint    ixsm;
    uint    ipcs;
    uint    tcpcs;

    eaddr   ra;       /* receive address */
    ulong   mta[128];       /* multicast table array */

    Rendez  rrendez;
    int rim;
    int rdfree;         /* rx descriptors awaiting packets */
    Rd* rdba;           /* receive descriptor base address */
    Block** rb;         /* receive buffers */
    int rdh;            /* receive descriptor head */
    int rdt;            /* receive descriptor tail */
    int rdtr;           /* receive delay timer ring value */

    Lock    tlock;
    int tdfree;
    Td* tdba;           /* transmit descriptor base address */
    Block** tb;         /* transmit buffers */
    int tdh;            /* transmit descriptor head */
    int tdt;            /* transmit descriptor tail */

    int txcw;
    int fcrtl;
    int fcrth;
};
```
Uses Nstatistics-550 271b.

⟨*macro* csr32r 279a⟩≡ (307b)
```
#define csr32r(c, r)    (*((c)->nic+((r)/4)))
```

⟨*macro* csr32w 279b⟩≡ (307b)
```
#define csr32w(c, r, v) (*((c)->nic+((r)/4)) = (v))
```

⟨*global* igbectlrhead 279c⟩≡ (307b)
```
static CtlrEtherIgbe* igbectlrhead;
```

⟨*global* igbectlrtail 279d⟩≡ (307b)
```
static CtlrEtherIgbe* igbectlrtail;
```

⟨*global* igberblock 280a⟩≡ (307b)

```
  static Lock igberblock;     /* free receive Blocks */
```

⟨*global* igberbpool 280b⟩≡ (307b)

```
  static Block* igberbpool;   /* receive Blocks for all igbe controllers */
```

⟨*global* statistics 280c⟩≡ (307b)

```
  static char* statistics[Nstatistics] = {
      "CRC Error",
      "Alignment Error",
      "Symbol Error",
      "RX Error",
      "Missed Packets",
      "Single Collision",
      "Excessive Collisions",
      "Multiple Collision",
      "Late Collisions",
      nil,
      "Collision",
      "Transmit Underrun",
      "Defer",
      "Transmit - No CRS",
      "Sequence Error",
      "Carrier Extension Error",
      "Receive Error Length",
      nil,
      "XON Received",
      "XON Transmitted",
      "XOFF Received",
      "XOFF Transmitted",
      "FC Received Unsupported",
      "Packets Received (64 Bytes)",
      "Packets Received (65-127 Bytes)",
      "Packets Received (128-255 Bytes)",
      "Packets Received (256-511 Bytes)",
      "Packets Received (512-1023 Bytes)",
      "Packets Received (1024-1522 Bytes)",
      "Good Packets Received",
      "Broadcast Packets Received",
      "Multicast Packets Received",
      "Good Packets Transmitted",
      nil,
      "Good Octets Received",
      nil,
      "Good Octets Transmitted",
      nil,
      nil,
      nil,
      "Receive No Buffers",
      "Receive Undersize",
      "Receive Fragment",
      "Receive Oversize",
      "Receive Jabber",
      nil,
      nil,
      nil,
      "Total Octets Received",
      nil,
      "Total Octets Transmitted",
      nil,
```

```
    "Total Packets Received",
    "Total Packets Transmitted",
    "Packets Transmitted (64 Bytes)",
    "Packets Transmitted (65-127 Bytes)",
    "Packets Transmitted (128-255 Bytes)",
    "Packets Transmitted (256-511 Bytes)",
    "Packets Transmitted (512-1023 Bytes)",
    "Packets Transmitted (1024-1522 Bytes)",
    "Multicast Packets Transmitted",
    "Broadcast Packets Transmitted",
    "TCP Segmentation Context Transmitted",
    "TCP Segmentation Context Fail",
};
```

Uses `Nstatistics-550` 271b.

⟨*function* `igbeifstat` 281⟩≡ (307b)

```
static long
igbeifstat(Ether* edev, void* a, long n, ulong offset)
{
    CtlrEtherIgbe *ctlr;
    char *p, *s;
    int i, l, r;
    uvlong tuvl, ruvl;

    ctlr = edev->ctlr;
    qlock(&ctlr->slock);
    p = malloc(READSTR);
    if(p == nil) {
        qunlock(&ctlr->slock);
        error(Enomem);
    }
    l = 0;
    for(i = 0; i < Nstatistics; i++){
        r = csr32r(ctlr, Statistics+i*4);
        if((s = statistics[i]) == nil)
            continue;
        switch(i){
        case Gorcl:
        case Gotcl:
        case Torl:
        case Totl:
            ruvl = r;
            ruvl += ((uvlong)csr32r(ctlr, Statistics+(i+1)*4))<<32;
            tuvl = ruvl;
            tuvl += ctlr->statistics[i];
            tuvl += ((uvlong)ctlr->statistics[i+1])<<32;
            if(tuvl == 0)
                continue;
            ctlr->statistics[i] = tuvl;
            ctlr->statistics[i+1] = tuvl>>32;
            l += snprint(p+l, READSTR-l, "%s: %llud %llud\n",
                s, tuvl, ruvl);
            i++;
            break;

        default:
            ctlr->statistics[i] += r;
            if(ctlr->statistics[i] == 0)
                continue;
            l += snprint(p+l, READSTR-l, "%s: %ud %ud\n",
```

```
                s, ctlr->statistics[i], r);
            break;
        }
    }

    l += snprint(p+l, READSTR-l, "lintr: %ud %ud\n",
        ctlr->lintr, ctlr->lsleep);
    l += snprint(p+l, READSTR-l, "rintr: %ud %ud\n",
        ctlr->rintr, ctlr->rsleep);
    l += snprint(p+l, READSTR-l, "tintr: %ud %ud\n",
        ctlr->tintr, ctlr->txdw);
    l += snprint(p+l, READSTR-l, "ixcs: %ud %ud %ud\n",
        ctlr->ixsm, ctlr->ipcs, ctlr->tcpcs);
    l += snprint(p+l, READSTR-l, "rdtr: %ud\n", ctlr->rdtr);
    l += snprint(p+l, READSTR-l, "Ctrlext: %08x\n", csr32r(ctlr, Ctrlext));

    l += snprint(p+l, READSTR-l, "eeprom:");
    for(i = 0; i < 0x40; i++){
        if(i && ((i & 0x07) == 0))
            l += snprint(p+l, READSTR-l, "\n       ");
        l += snprint(p+l, READSTR-l, " %4.4uX", ctlr->eeprom[i]);
    }
    l += snprint(p+l, READSTR-l, "\n");

    if(ctlr->mii != nil && ctlr->mii->curphy != nil){
        l += snprint(p+l, READSTR-l, "phy:   ");
        for(i = 0; i < NMiiPhyr; i++){
            if(i && ((i & 0x07) == 0))
                l += snprint(p+l, READSTR-l, "\n       ");
            r = miimir(ctlr->mii, i);
            l += snprint(p+l, READSTR-l, " %4.4uX", r);
        }
        snprint(p+l, READSTR-l, "\n");
    }
    n = readstr(offset, a, n, p);
    free(p);
    qunlock(&ctlr->slock);

    return n;
}
```

Uses Ctrlext-498 271b, Gorcl-546 271b, Gotcl-547 271b, NMiiPhyr 314b, Nstatistics-550 271b, Statistics-545 271b, Torl-548 271b, Totl-549 271b, csr32r-786 279a, miimir() 311a, readstr(), and statistics-792 280c.

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)25 282a⟩≡ (307b)

```
enum {
    CMrdtr,
};
```

⟨*global* igbectlmsg 282b⟩≡ (307b)

```
static Cmdtab igbectlmsg[] = {
    CMrdtr, "rdtr", 2,
};
```

Uses CMrdtr-793 282a.

⟨*function* igbectl 282c⟩≡ (307b)

```
static long
igbectl(Ether* edev, void* buf, long n)
{
    int v;
    char *p;
```

```
    CtlrEtherIgbe *ctlr;
    Cmdbuf *cb;
    Cmdtab *ct;

    if((ctlr = edev->ctlr) == nil)
        error(Enonexist);

    cb = parsecmd(buf, n);
    if(waserror()){
        free(cb);
        nexterror();
    }

    ct = lookupcmd(cb, igbectlmsg, nelem(igbectlmsg));
    switch(ct->index){
    case CMrdtr:
        v = strtol(cb->f[1], &p, 0);
        if(v < 0 || p == cb->f[1] || v > 0xFFFF)
            error(Ebadarg);
        ctlr->rdtr = v;
        csr32w(ctlr, Rdtr, Fpd|v);
        break;
    }
    free(cb);
    poperror();

    return n;
}
```

Uses `CMrdtr-793` 282a, `Fpd-738` 277b, `Rdtr-527` 271b, `csr32w-787` 279b, `igbectlmsg-794` 282b, `lookupcmd()`, and `parsecmd()`.

⟨*function* `igbepromiscuous` 283a⟩≡ (307b)

```
static void
igbepromiscuous(void* arg, int on)
{
    int rctl;
    CtlrEtherIgbe *ctlr;
    Ether *edev;

    edev = arg;
    ctlr = edev->ctlr;

    rctl = csr32r(ctlr, Rctl);
    rctl &= ~MoMASK;
    rctl |= Mo47b36;
    if(on)
        rctl |= Upe|Mpe;
    else
        rctl &= ~(Upe|Mpe);
    csr32w(ctlr, Rctl, rctl|Mpe);   /* temporarily keep Mpe on */
}
```

Uses `Mo47b36-693` 275d, `MoMASK-692` 275d, `Mpe-681` 275d, `Rctl-507` 271b, `Upe-680` 275d, `csr32r-786` 279a, and `csr32w-787` 279b.

⟨*function* `igbemulticast` 283b⟩≡ (307b)

```
static void
igbemulticast(void* arg, uchar* addr, int add)
{
    int bit, x;
    CtlrEtherIgbe *ctlr;
    Ether *edev;
```

```
    edev = arg;
    ctlr = edev->ctlr;

    x = addr[5]>>1;
    bit = ((addr[5] & 1)<<4)|(addr[4]>>4);
    /*
     * multiple ether addresses can hash to the same filter bit,
     * so it's never safe to clear a filter bit.
     * if we want to clear filter bits, we need to keep track of
     * all the multicast addresses in use, clear all the filter bits,
     * then set the ones corresponding to in-use addresses.
     */
    if(add)
        ctlr->mta[x] |= 1<<bit;
//  else
//      ctlr->mta[x] &= ~(1<<bit);

    csr32w(ctlr, Mta+x*4, ctlr->mta[x]);
}
```

Uses `Mta-552` 271b and `csr32w-787` 279b.

⟨*function* `igberballoc` 284a⟩≡ (307b)

```
static Block*
igberballoc(void)
{
    Block *bp;

    ilock(&igberblock);
    if((bp = igberbpool) != nil){
        igberbpool = bp->next;
        bp->next = nil;
        arch_xinc(&bp->ref);    /* prevent bp from being freed */
    }
    iunlock(&igberblock);

    return bp;
}
```

Uses `igberblock-790` 280a and `igberbpool-791` 280b.

⟨*function* `igberbfree` 284b⟩≡ (307b)

```
static void
igberbfree(Block* bp)
{
    bp->rp = bp->lim - Rbsz;
    bp->wp = bp->rp;
    bp->flag &= ~(Bipck | Budpck | Btcpck | Bpktck);

    ilock(&igberblock);
    bp->next = igberbpool;
    igberbpool = bp;
    iunlock(&igberblock);
}
```

Uses `Rbsz-785` 278c, `igberblock-790` 280a, and `igberbpool-791` 280b.

⟨*function* `igbeim` 284c⟩≡ (307b)

```
static void
igbeim(CtlrEtherIgbe* ctlr, int im)
{
    ilock(&ctlr->imlock);
```

```
    ctlr->im |= im;
    csr32w(ctlr, Ims, ctlr->im);
    iunlock(&ctlr->imlock);
}
```

Uses Ims-505 271b and csr32w-787 279b.

⟨*function* igbelim 285a⟩≡ (307b)

```
static int
igbelim(void* ctlr)
{
    return ((CtlrEtherIgbe*)ctlr)->lim != 0;
}
```

⟨*function* igbelproc 285b⟩≡ (307b)

```
static void
igbelproc(void* arg)
{
    CtlrEtherIgbe *ctlr;
    Ether *edev;
    MiiPhy *phy;
    int ctrl, r;

    edev = arg;
    ctlr = edev->ctlr;
    for(;;){
        if(ctlr->mii == nil || ctlr->mii->curphy == nil) {
            sched();
            continue;
        }

        /*
         * To do:
         *  logic to manage status change,
         *  this is incomplete but should work
         *  one time to set up the hardware.
         *
         *  MiiPhy.speed, etc. should be in Mii.
         */
        if(miistatus(ctlr->mii) < 0)
            //continue;
            goto enable;

        phy = ctlr->mii->curphy;
        ctrl = csr32r(ctlr, Ctrl);

        switch(ctlr->id){
        case i82543gc:
        case i82544ei:
        case i82544eif:
        default:
            if(!(ctrl & Asde)){
                ctrl &= ~(SspeedMASK|Ilos|Fd);
                ctrl |= Frcdplx|Frcspd;
                if(phy->speed == 1000)
                    ctrl |= Sspeed1000;
                else if(phy->speed == 100)
                    ctrl |= Sspeed100;
                if(phy->fd)
                    ctrl |= Fd;
            }
```

```
            break;

        case i82540em:
        case i82540eplp:
        case i82547gi:
        case i82541gi:
        case i82541gi2:
        case i82541pi:
            break;
        }

        /*
         * Collision Distance.
         */
        r = csr32r(ctlr, Tctl);
        r &= ~ColdMASK;
        if(phy->fd)
            r |= 64<<ColdSHIFT;
        else
            r |= 512<<ColdSHIFT;
        csr32w(ctlr, Tctl, r);

        /*
         * Flow control.
         */
        if(phy->rfc)
            ctrl |= Rfce;
        if(phy->tfc)
            ctrl |= Tfce;
        csr32w(ctlr, Ctrl, ctrl);

enable:
        ctlr->lim = 0;
        igbeim(ctlr, Lsc);

        ctlr->lsleep++;
        sleep(&ctlr->lrendez, igbelim, ctlr);
    }
}
```

Uses `Asde-559` 272, `ColdMASK-716` 276a, `ColdSHIFT-717` 276a, `Ctrl-494` 271b, `Fd-595` 273b, `Frcdplx-568` 272, `Frcspd-567` 272, `Ilos-561` 272, `Lsc-642` 274d, `Rfce-574` 272, `Sspeed100-565` 272, `Sspeed1000-566` 272, `SspeedMASK-562` 272, `Tctl-511` 271b, `Tfce-575` 272, `csr32r-786` 279a, `csr32w-787` 279b, `i82540em-482` 271a, `i82540eplp-483` 271a, `i82541gi-489` 271a, `i82541gi2-490` 271a, `i82541pi-491` 271a, `i82543gc-478` 271a, `i82544ei-479` 271a, `i82544eif-480` 271a, `i82547gi-487` 271a, `igbeim()` 284c, `igbelim()` 285a, and `miistatus()` 313.

⟨*function* `igbetxinit` 286⟩≡ (307b)

```
static void
igbetxinit(CtlrEtherIgbe* ctlr)
{
    int i, r;
    Block *bp;

    csr32w(ctlr, Tctl, (0x0F<<CtSHIFT)|Psp|(66<<ColdSHIFT));
    switch(ctlr->id){
    default:
        r = 6;
        break;
    case i82543gc:
    case i82544ei:
    case i82544eif:
```

```
case i82544gc:
case i82540em:
case i82540eplp:
case i82541ei:
case i82541gi:
case i82541gi2:
case i82541pi:
case i82545em:
case i82545gmc:
case i82546gb:
case i82546eb:
case i82547ei:
case i82547gi:
    r = 8;
    break;
}
csr32w(ctlr, Tipg, (6<<20)|(8<<10)|r);
csr32w(ctlr, Ait, 0);
csr32w(ctlr, Txdmac, 0);

csr32w(ctlr, Tdbal, PCIWADDR(ctlr->tdba));
csr32w(ctlr, Tdbah, 0);
csr32w(ctlr, Tdlen, ctlr->ntd*sizeof(Td));
ctlr->tdh = PREV(0, ctlr->ntd);
csr32w(ctlr, Tdh, 0);
ctlr->tdt = 0;
csr32w(ctlr, Tdt, 0);

for(i = 0; i < ctlr->ntd; i++){
    if((bp = ctlr->tb[i]) != nil){
        ctlr->tb[i] = nil;
        freeb(bp);
    }
    memset(&ctlr->tdba[i], 0, sizeof(Td));
}
ctlr->tdfree = ctlr->ntd;

csr32w(ctlr, Tidv, 128);
r = (4<<WthreshSHIFT)|(4<<HthreshSHIFT)|(8<<PthreshSHIFT);

switch(ctlr->id){
default:
    break;
case i82540em:
case i82540eplp:
case i82547gi:
case i82545em:
case i82545gmc:
case i82546gb:
case i82546eb:
case i82541gi:
case i82541gi2:
case i82541pi:
    r = csr32r(ctlr, Txdctl);
    r &= ~WthreshMASK;
    r |= Gran|(4<<WthreshSHIFT);

    csr32w(ctlr, Tadv, 64);
    break;
}
```

```
    csr32w(ctlr, Txdctl, r);

    r = csr32r(ctlr, Tctl);
    r |= Ten;
    csr32w(ctlr, Tctl, r);
}
```

Uses Ait-514 271b, ColdSHIFT-717 276a, CtSHIFT-715 276a, Gran-728 276b, HthreshSHIFT-725 276b, PREV 255c, Psp-713 276a, PthreshSHIFT-723 276b, Tadv-544 271b, Tctl-511 271b, Tdbah-538 271b, Tdbal-537 271b, Tdh-540 271b, Tdlen-539 271b, Tdt-541 271b, Ten-712 276a, Tidv-542 271b, Tipg-512 271b, Txdctl-543 271b, Txdmac-530 271b, WthreshMASK-726 276b, WthreshSHIFT-727 276b, csr32r-786 279a, csr32w-787 279b, i82540em-482 271a, i82540eplp-483 271a, i82541ei-488 271a, i82541gi-489 271a, i82541gi2-490 271a, i82541pi-491 271a, i82543gc-478 271a, i82544ei-479 271a, i82544eif-480 271a, i82544gc-481 271a, i82545em-484 271a, i82545gmc-485 271a, i82546eb-493 271a, i82546gb-492 271a, i82547ei-486 271a, and i82547gi-487 271a.

⟨*function* igbetransmit 288⟩≡ (307b)

```
static void
igbetransmit(Ether* edev)
{
    Td *td;
    Block *bp;
    CtlrEtherIgbe *ctlr;
    int tdh, tdt;

    ctlr = edev->ctlr;

    ilock(&ctlr->tlock);

    /*
     * Free any completed packets
     */
    tdh = ctlr->tdh;
    while(NEXT(tdh, ctlr->ntd) != csr32r(ctlr, Tdh)){
        if((bp = ctlr->tb[tdh]) != nil){
            ctlr->tb[tdh] = nil;
            freeb(bp);
        }
        memset(&ctlr->tdba[tdh], 0, sizeof(Td));
        tdh = NEXT(tdh, ctlr->ntd);
    }
    ctlr->tdh = tdh;

    /*
     * Try to fill the ring back up.
     */
    tdt = ctlr->tdt;
    while(NEXT(tdt, ctlr->ntd) != tdh){
        if((bp = qget(edev->oq)) == nil)
            break;
        td = &ctlr->tdba[tdt];
        td->addr[0] = PCIWADDR(bp->rp);
        td->control = ((BLEN(bp) & LenMASK)<<LenSHIFT);
        td->control |= Dext|Ifcs|Teop|DtypeDD;
        ctlr->tb[tdt] = bp;
        tdt = NEXT(tdt, ctlr->ntd);
        if(NEXT(tdt, ctlr->ntd) == tdh){
            td->control |= Rs;
            ctlr->txdw++;
            ctlr->tdt = tdt;
            csr32w(ctlr, Tdt, tdt);
```

```
            igbeim(ctlr, Txdw);
            break;
        }
        ctlr->tdt = tdt;
        csr32w(ctlr, Tdt, tdt);
    }

    iunlock(&ctlr->tlock);
}
```

Uses Dext-764 278a, DtypeDD-756 278a, Ifcs-760 278a, LenMASK-753 278a, LenSHIFT-754 278a, NEXT 255b, Rs-762 278a, Tdh-540 271b, Tdt-541 271b, Teop-758 278a, Txdw-640 274d, csr32r-786 279a, csr32w-787 279b, and igbeim() 284c.

⟨*function* igbereplenish 289a⟩≡ (307b)

```
static void
igbereplenish(CtlrEtherIgbe* ctlr)
{
    Rd *rd;
    int rdt;
    Block *bp;

    rdt = ctlr->rdt;
    while(NEXT(rdt, ctlr->nrd) != ctlr->rdh){
        rd = &ctlr->rdba[rdt];
        if(ctlr->rb[rdt] == nil){
            bp = igberballoc();
            if(bp == nil){
                iprint("#l%d: igbereplenish: no available buffers\n",
                    ctlr->edev->ctlrno);
                break;
            }
            ctlr->rb[rdt] = bp;
            rd->addr[0] = PCIWADDR(bp->rp);
            rd->addr[1] = 0;
        }
        arch_coherence();
        rd->status = 0;
        rdt = NEXT(rdt, ctlr->nrd);
        ctlr->rdfree++;
    }
    ctlr->rdt = rdt;
    csr32w(ctlr, Rdt, rdt);
}
```

Uses NEXT 255b, Rdt-526 271b, csr32w-787 279b, and igberballoc() 284a.

⟨*function* igberxinit 289b⟩≡ (307b)

```
static void
igberxinit(CtlrEtherIgbe* ctlr)
{
    int i;
    Block *bp;

    /* temporarily keep Mpe on */
    csr32w(ctlr, Rctl, Dpf|Bsize2048|Bam|RdtmsHALF|Mpe);

    csr32w(ctlr, Rdbal, PCIWADDR(ctlr->rdba));
    csr32w(ctlr, Rdbah, 0);
    csr32w(ctlr, Rdlen, ctlr->nrd*sizeof(Rd));
    ctlr->rdh = 0;
    csr32w(ctlr, Rdh, 0);
    ctlr->rdt = 0;
```

```
    csr32w(ctlr, Rdt, 0);
    ctlr->rdtr = 0;
    csr32w(ctlr, Rdtr, Fpd|0);

    for(i = 0; i < ctlr->nrd; i++){
        if((bp = ctlr->rb[i]) != nil){
            ctlr->rb[i] = nil;
            freeb(bp);
        }
    }
    igbereplenish(ctlr);

    switch(ctlr->id){
    case i82540em:
    case i82540eplp:
    case i82541gi:
    case i82541gi2:
    case i82541pi:
    case i82545em:
    case i82545gmc:
    case i82546gb:
    case i82546eb:
    case i82547gi:
        csr32w(ctlr, Radv, 64);
        break;
    }
    csr32w(ctlr, Rxdctl, (8<<WthreshSHIFT)|(8<<HthreshSHIFT)|4);

    /*
     * Disable checksum offload as it has known bugs.
     */
    csr32w(ctlr, Rxcsum, ETHERHDRSIZE<<PcssSHIFT);
}
```

Uses Bam-697 275d, Bsize2048-699 275d, Dpf-707 275d, ETHERHDRSIZE 240f, Fpd-738 277b, HthreshSHIFT-725 276b, Mpe-681 275d, PcssSHIFT-732 276c, Radv-529 271b, Rctl-507 271b, Rdbah-523 271b, Rdbal-522 271b, Rdh-525 271b, Rdlen-524 271b, Rdt-526 271b, RdtmsHALF-689 275d, Rdtr-527 271b, Rxcsum-551 271b, Rxdctl-528 271b, WthreshSHIFT-727 276b, csr32w-787 279b, i82540em-482 271a, i82540eplp-483 271a, i82541gi-489 271a, i82541gi2-490 271a, i82541pi-491 271a, i82545em-484 271a, i82545gmc-485 271a, i82546eb-493 271a, i82546gb-492 271a, i82547gi-487 271a, and igbereplenish() 289a.

⟨*function* igberim 290a⟩≡ (307b)

```
static int
igberim(void* ctlr)
{
    return ((CtlrEtherIgbe*)ctlr)->rim != 0;
}
```

⟨*function* igberproc 290b⟩≡ (307b)

```
static void
igberproc(void* arg)
{
    Rd *rd;
    Block *bp;
    CtlrEtherIgbe *ctlr;
    int r, rdh;
    Ether *edev;

    edev = arg;
    ctlr = edev->ctlr;
```

```
igberxinit(ctlr);
r = csr32r(ctlr, Rctl);
r |= Ren;
csr32w(ctlr, Rctl, r);

for(;;){
    ctlr->rim = 0;
    igbeim(ctlr, Rxt0|Rxo|Rxdmt0|Rxseq);
    ctlr->rsleep++;
    sleep(&ctlr->rrendez, igberim, ctlr);

    rdh = ctlr->rdh;
    for(;;){
        rd = &ctlr->rdba[rdh];

        if(!(rd->status & Rdd))
            break;

        /*
         * Accept eop packets with no errors.
         * With no errors and the Ixsm bit set,
         * the descriptor status Tpcs and Ipcs bits give
         * an indication of whether the checksums were
         * calculated and valid.
         */
        if((rd->status & Reop) && rd->errors == 0){
            bp = ctlr->rb[rdh];
            ctlr->rb[rdh] = nil;
            bp->wp += rd->length;
            bp->next = nil;
            if(!(rd->status & Ixsm)){
                ctlr->ixsm++;
                if(rd->status & Ipcs){
                    /*
                     * IP checksum calculated
                     * (and valid as errors == 0).
                     */
                    ctlr->ipcs++;
                    bp->flag |= Bipck;
                }
                if(rd->status & Tcpcs){
                    /*
                     * TCP/UDP checksum calculated
                     * (and valid as errors == 0).
                     */
                    ctlr->tcpcs++;
                    bp->flag |= Btcpck|Budpck;
                }
                bp->checksum = rd->checksum;
                bp->flag |= Bpktck;
            }
            etheriq(edev, bp, 1);
        }
        else if(ctlr->rb[rdh] != nil){
            freeb(ctlr->rb[rdh]);
            ctlr->rb[rdh] = nil;
        }

        memset(rd, 0, sizeof(Rd));
        arch_coherence();
```

```
            ctlr->rdfree--;
            rdh = NEXT(rdh, ctlr->nrd);
        }
        ctlr->rdh = rdh;

        if(ctlr->rdfree < ctlr->nrd/2 || (ctlr->rim & Rxdmt0))
            igbereplenish(ctlr);
    }
}
```

Uses Ipcs-744 277d, Ixsm-741 277d, NEXT 255b, Rctl-507 271b, Rdd-739 277d, Ren-678 275d, Reop-740 277d, Rxdmt0-644 274d, Rxo-645 274d, Rxseq-643 274d, Rxt0-646 274d, Tcpcs-743 277d, csr32r-786 279a, csr32w-787 279b, igbeim() 284c, igbereplenish() 289a, igberim() 290a, and igberxinit() 289b.

⟨*function* igbeattach 292⟩≡ (307b)

```
static void
igbeattach(Ether* edev)
{
    Block *bp;
    CtlrEtherIgbe *ctlr;
    char name[KNAMELEN];

    ctlr = edev->ctlr;
    ctlr->edev = edev;          /* point back to Ether* */
    qlock(&ctlr->alock);
    if(ctlr->alloc != nil){         /* already allocated? */
        qunlock(&ctlr->alock);
        return;
    }

    ctlr->tb = nil;
    ctlr->rb = nil;
    ctlr->alloc = nil;
    ctlr->nrb = 0;
    if(waserror()){
        while(ctlr->nrb > 0){
            bp = igberballoc();
            bp->free = nil;
            freeb(bp);
            ctlr->nrb--;
        }
        free(ctlr->tb);
        ctlr->tb = nil;
        free(ctlr->rb);
        ctlr->rb = nil;
        free(ctlr->alloc);
        ctlr->alloc = nil;
        qunlock(&ctlr->alock);
        nexterror();
    }

    ctlr->nrd = ROUND(Nrd, 8);
    ctlr->ntd = ROUND(Ntd, 8);
    ctlr->alloc = malloc(ctlr->nrd*sizeof(Rd)+ctlr->ntd*sizeof(Td) + 127);
    if(ctlr->alloc == nil) {
        print("igbe: can't allocate ctlr->alloc\n");
        error(Enomem);
    }
    ctlr->rdba = (Rd*)ROUNDUP((uintptr)ctlr->alloc, 128);
    ctlr->tdba = (Td*)(ctlr->rdba+ctlr->nrd);
```

```
    ctlr->rb = malloc(ctlr->nrd*sizeof(Block*));
    ctlr->tb = malloc(ctlr->ntd*sizeof(Block*));
    if (ctlr->rb == nil || ctlr->tb == nil) {
        print("igbe: can't allocate ctlr->rb or ctlr->tb\n");
        error(Enomem);
    }

    for(ctlr->nrb = 0; ctlr->nrb < Nrb; ctlr->nrb++){
        if((bp = allocb(Rbsz)) == nil)
            break;
        bp->free = igberbfree;
        freeb(bp);
    }

    snprint(name, KNAMELEN, "#l%dlproc", edev->ctlrno);
    kproc(name, igbelproc, edev);

    snprint(name, KNAMELEN, "#l%drproc", edev->ctlrno);
    kproc(name, igberproc, edev);

    igbetxinit(ctlr);

    qunlock(&ctlr->alock);
    poperror();
}
```

Uses `Nrb-784` 278c, `Nrd-782` 278c, `Ntd-783` 278c, `Rbsz-785` 278c, `igbelproc()` 285b, `igberballoc()` 284a, `igberbfree()` 284b, `igberproc()` 290b, and `igbetxinit()` 286.

⟨*function* `igbeinterrupt` 293⟩≡ (307b)

```
static void
igbeinterrupt(Ureg*, void* arg)
{
    CtlrEtherIgbe *ctlr;
    Ether *edev;
    int icr, im, txdw;

    edev = arg;
    ctlr = edev->ctlr;

    ilock(&ctlr->imlock);
    csr32w(ctlr, Imc, ~0);
    im = ctlr->im;
    txdw = 0;

    while((icr = csr32r(ctlr, Icr) & ctlr->im) != 0){
        if(icr & Lsc){
            im &= ~Lsc;
            ctlr->lim = icr & Lsc;
            wakeup(&ctlr->lrendez);
            ctlr->lintr++;
        }
        if(icr & (Rxt0|Rxo|Rxdmt0|Rxseq)){
            im &= ~(Rxt0|Rxo|Rxdmt0|Rxseq);
            ctlr->rim = icr & (Rxt0|Rxo|Rxdmt0|Rxseq);
            wakeup(&ctlr->rrendez);
            ctlr->rintr++;
        }
        if(icr & Txdw){
            im &= ~Txdw;
            txdw++;
```

```
            ctlr->tintr++;
        }
    }

    ctlr->im = im;
    csr32w(ctlr, Ims, im);
    iunlock(&ctlr->imlock);

    if(txdw)
        igbetransmit(edev);
}
```

Uses Icr-503 271b, Imc-506 271b, Ims-505 271b, Lsc-642 274d, Rxdmt0-644 274d, Rxo-645 274d, Rxseq-643 274d, Rxt0-646 274d, Txdw-640 274d, csr32r-786 279a, csr32w-787 279b, and igbetransmit() 288.

⟨*function* i82543mdior 294a⟩≡ (307b)

```
static int
i82543mdior(CtlrEtherIgbe* ctlr, int n)
{
    int ctrl, data, i, r;

    /*
     * Read n bits from the Management Data I/O Interface.
     */
    ctrl = csr32r(ctlr, Ctrl);
    r = (ctrl & ~Mddo)|Mdco;
    data = 0;
    for(i = n-1; i >= 0; i--){
        if(csr32r(ctlr, Ctrl) & Mdd)
            data |= (1<<i);
        csr32w(ctlr, Ctrl, Mdc|r);
        csr32w(ctlr, Ctrl, r);
    }
    csr32w(ctlr, Ctrl, ctrl);

    return data;
}
```

Uses Ctrl-494 271b, Mdc-655 275a, Mdco-656 275a, Mdd-653 275a, Mddo-654 275a, csr32r-786 279a, and csr32w-787 279b.

⟨*function* i82543mdiow 294b⟩≡ (307b)

```
static int
i82543mdiow(CtlrEtherIgbe* ctlr, int bits, int n)
{
    int ctrl, i, r;

    /*
     * Write n bits to the Management Data I/O Interface.
     */
    ctrl = csr32r(ctlr, Ctrl);
    r = Mdco|Mddo|ctrl;
    for(i = n-1; i >= 0; i--){
        if(bits & (1<<i))
            r |= Mdd;
        else
            r &= ~Mdd;
        csr32w(ctlr, Ctrl, Mdc|r);
        csr32w(ctlr, Ctrl, r);
    }
    csr32w(ctlr, Ctrl, ctrl);

    return 0;
```

```
}
```
Uses `Ctrl-494` 271b, `Mdc-655` 275a, `Mdco-656` 275a, `Mdd-653` 275a, `Mddo-654` 275a, `csr32r-786` 279a, and `csr32w-787` 279b.

⟨*function* `i82543miimir` 295a⟩≡ (307b)
```
static int
i82543miimir(Mii* mii, int pa, int ra)
{
    int data;
    CtlrEtherIgbe *ctlr;

    ctlr = mii->ctlr;

    /*
     * MII Management Interface Read.
     *
     * Preamble;
     * ST+OP+PHYAD+REGAD;
     * TA + 16 data bits.
     */
    i82543mdiow(ctlr, 0xFFFFFFFF, 32);
    i82543mdiow(ctlr, 0x1800|(pa<<5)|ra, 14);
    data = i82543mdior(ctlr, 18);

    if(data & 0x10000)
        return -1;

    return data & 0xFFFF;
}
```
Uses `i82543mdior()` 294a and `i82543mdiow()` 294b.

⟨*function* `i82543miimiw` 295b⟩≡ (307b)
```
static int
i82543miimiw(Mii* mii, int pa, int ra, int data)
{
    CtlrEtherIgbe *ctlr;

    ctlr = mii->ctlr;

    /*
     * MII Management Interface Write.
     *
     * Preamble;
     * ST+OP+PHYAD+REGAD+TA + 16 data bits;
     * Z.
     */
    i82543mdiow(ctlr, 0xFFFFFFFF, 32);
    data &= 0xFFFF;
    data |= (0x05<<(5+5+2+16))|(pa<<(5+2+16))|(ra<<(2+16))|(0x02<<16);
    i82543mdiow(ctlr, data, 32);

    return 0;
}
```
Uses `i82543mdiow()` 294b.

⟨*function* `igbemiimir` 295c⟩≡ (307b)
```
static int
igbemiimir(Mii* mii, int pa, int ra)
{
    CtlrEtherIgbe *ctlr;
```

```
    int mdic, timo;

    ctlr = mii->ctlr;

    csr32w(ctlr, Mdic, MDIrop|(pa<<MDIpSHIFT)|(ra<<MDIrSHIFT));
    mdic = 0;
    for(timo = 64; timo; timo--){
        mdic = csr32r(ctlr, Mdic);
        if(mdic & (MDIe|MDIready))
            break;
        arch_microdelay(1);
    }

    if((mdic & (MDIe|MDIready)) == MDIready)
        return mdic & 0xFFFF;
    return -1;
}
```

Uses MDIe-639 274c, MDIpSHIFT-634 274c, MDIrSHIFT-632 274c, MDIready-637 274c, MDIrop-636 274c, Mdic-499 271b, csr32r-786 279a, and csr32w-787 279b.

⟨*function* igbemiimiw 296a⟩≡ (307b)

```
static int
igbemiimiw(Mii* mii, int pa, int ra, int data)
{
    CtlrEtherIgbe *ctlr;
    int mdic, timo;

    ctlr = mii->ctlr;

    data &= MDIdMASK;
    csr32w(ctlr, Mdic, MDIwop|(pa<<MDIpSHIFT)|(ra<<MDIrSHIFT)|data);
    mdic = 0;
    for(timo = 64; timo; timo--){
        mdic = csr32r(ctlr, Mdic);
        if(mdic & (MDIe|MDIready))
            break;
        arch_microdelay(1);
    }
    if((mdic & (MDIe|MDIready)) == MDIready)
        return 0;
    return -1;
}
```

Uses MDIdMASK-629 274c, MDIe-639 274c, MDIpSHIFT-634 274c, MDIrSHIFT-632 274c, MDIready-637 274c, MDIwop-635 274c, Mdic-499 271b, csr32r-786 279a, and csr32w-787 279b.

⟨*function* igbemii 296b⟩≡ (307b)

```
static int
igbemii(CtlrEtherIgbe* ctlr)
{
    MiiPhy *phy;
    int ctrl, p, r;

    r = csr32r(ctlr, Status);
    if(r & Tbimode)
        return -1;
    if((ctlr->mii = malloc(sizeof(Mii))) == nil)
        return -1;
    ctlr->mii->ctlr = ctlr;

    ctrl = csr32r(ctlr, Ctrl);
```

```
ctrl |= Slu;

switch(ctlr->id){
case i82543gc:
    ctrl |= Frcdplx|Frcspd;
    csr32w(ctlr, Ctrl, ctrl);

    /*
     * The reset pin direction (Mdro) should already
     * be set from the EEPROM load.
     * If it's not set this configuration is unexpected
     * so bail.
     */
    r = csr32r(ctlr, Ctrlext);
    if(!(r & Mdro)) {
        print("igbe: 82543gc Mdro not set\n");
        return -1;
    }
    csr32w(ctlr, Ctrlext, r);
    arch_delay(20);
    r = csr32r(ctlr, Ctrlext);
    r &= ~Mdr;
    csr32w(ctlr, Ctrlext, r);
    arch_delay(20);
    r = csr32r(ctlr, Ctrlext);
    r |= Mdr;
    csr32w(ctlr, Ctrlext, r);
    arch_delay(20);

    ctlr->mii->mir = i82543miimir;
    ctlr->mii->miw = i82543miimiw;
    break;
case i82544ei:
case i82544eif:
case i82544gc:
case i82540em:
case i82540eplp:
case i82547ei:
case i82547gi:
case i82541ei:
case i82541gi:
case i82541gi2:
case i82541pi:
case i82545em:
case i82545gmc:
case i82546gb:
case i82546eb:
    ctrl &= ~(Frcdplx|Frcspd);
    csr32w(ctlr, Ctrl, ctrl);
    ctlr->mii->mir = igbemiimir;
    ctlr->mii->miw = igbemiimiw;
    break;
default:
    free(ctlr->mii);
    ctlr->mii = nil;
    return -1;
}

if(mii(ctlr->mii, ~0) == 0 || (phy = ctlr->mii->curphy) == nil){
    free(ctlr->mii);
```

```
        ctlr->mii = nil;
        return -1;
    }
    USED(phy);
    // print("oui %X phyno %d\n", phy->oui, phy->phyno);

    /*
     * 8254X-specific PHY registers not in 802.3:
     *  0x10    PHY specific control
     *  0x14    extended PHY specific control
     * Set appropriate values then reset the PHY to have
     * changes noted.
     */
    switch(ctlr->id){
    case i82547gi:
    case i82541gi:
    case i82541gi2:
    case i82541pi:
    case i82545em:
    case i82545gmc:
    case i82546gb:
    case i82546eb:
        break;
    default:
        r = miimir(ctlr->mii, 16);
        r |= 0x0800;            /* assert CRS on Tx */
        r |= 0x0060;            /* auto-crossover all speeds */
        r |= 0x0002;            /* polarity reversal enabled */
        miimiw(ctlr->mii, 16, r);

        r = miimir(ctlr->mii, 20);
        r |= 0x0070;            /* +25MHz clock */
        r &= ~0x0F00;
        r |= 0x0100;            /* 1x downshift */
        miimiw(ctlr->mii, 20, r);

        miireset(ctlr->mii);
        p = 0;
        if(ctlr->txcw & TxcwPs)
            p |= AnaP;
        if(ctlr->txcw & TxcwAs)
            p |= AnaAP;
        miiane(ctlr->mii, ~0, p, ~0);
        break;
    }
    return 0;
}
```

Uses `AnaAP` 315c, `AnaP` 315c, `Ctrl-494` 271b, `Ctrlext-498` 271b, `Frcdplx-568` 272, `Frcspd-567` 272, `Mdr-657` 275a, `Mdro-658` 275a, `Slu-560` 272, `Status-496` 271b, `Tbimode-580` 273a, `TxcwAs-664` 275b, `TxcwPs-663` 275b, `csr32r-786` 279a, `csr32w-787` 279b, `i82540em-482` 271a, `i82540eplp-483` 271a, `i82541ei-488` 271a, `i82541gi-489` 271a, `i82541gi2-490` 271a, `i82541pi-491` 271a, `i82543gc-478` 271a, `i82543miimir()` 295a, `i82543miimiw()` 295b, `i82544ei-479` 271a, `i82544eif-480` 271a, `i82544gc-481` 271a, `i82545em-484` 271a, `i82545gmc-485` 271a, `i82546eb-493` 271a, `i82546gb-492` 271a, `i82547ei-486` 271a, `i82547gi-487` 271a, `igbemiimir()` 295c, `igbemiimiw()` 296a, `mii()` 310, `miiane()` 311d, `miimir()` 311a, `miimiw()` 311b, and `miireset()` 311c.

⟨*function* `at93c46io` 298⟩≡ (307b)

```
static int
at93c46io(CtlrEtherIgbe* ctlr, char* op, int data)
{
    char *lp, *p;
```

```
int i, loop, eecd, r;

eecd = csr32r(ctlr, Eecd);

r = 0;
loop = -1;
lp = nil;
for(p = op; *p != '\0'; p++){
    switch(*p){
    default:
        return -1;
    case ' ':
        continue;
    case ':':           /* start of loop */
        loop = strtol(p+1, &lp, 0)-1;
        lp--;
        if(p == lp)
            loop = 7;
        p = lp;
        continue;
    case ';':           /* end of loop */
        if(lp == nil)
            return -1;
        loop--;
        if(loop >= 0)
            p = lp;
        else
            lp = nil;
        continue;
    case 'C':           /* assert clock */
        eecd |= Sk;
        break;
    case 'c':           /* deassert clock */
        eecd &= ~Sk;
        break;
    case 'D':           /* next bit in 'data' byte */
        if(loop < 0)
            return -1;
        if(data & (1<<loop))
            eecd |= Di;
        else
            eecd &= ~Di;
        break;
    case 'O':           /* collect data output */
        i = (csr32r(ctlr, Eecd) & Do) != 0;
        if(loop >= 0)
            r |= (i<<loop);
        else
            r = i;
        continue;
    case 'I':           /* assert data input */
        eecd |= Di;
        break;
    case 'i':           /* deassert data input */
        eecd &= ~Di;
        break;
    case 'S':           /* enable chip select */
        eecd |= Cs;
        break;
    case 's':           /* disable chip select */
```

```
            eecd &= ~Cs;
            break;
        }
        csr32w(ctlr, Eecd, eecd);
        arch_microdelay(50);
    }
    if(loop >= 0)
        return -1;
    return r;
}
```

Uses Cs-602 273c, Di-603 273c, Do-604 273c, Eecd-497 271b, Sk-601 273c, csr32r-786 279a, and csr32w-787 279b.

⟨*function* at93c46r 300⟩≡ (307b)

```
static int
at93c46r(CtlrEtherIgbe* ctlr)
{
    ushort sum;
    char rop[20];
    int addr, areq, bits, data, eecd, i;

    eecd = csr32r(ctlr, Eecd);
    if(eecd & Spi){
        print("igbe: SPI EEPROM access not implemented\n");
        return 0;
    }
    if(eecd & (Eeszaddr|Eesz256))
        bits = 8;
    else
        bits = 6;

    sum = 0;

    switch(ctlr->id){
    default:
        areq = 0;
        break;
    case i82540em:
    case i82540eplp:
    case i82541ei:
    case i82541gi:
    case i82541gi2:
    case i82541pi:
    case i82545em:
    case i82545gmc:
    case i82546gb:
    case i82546eb:
    case i82547ei:
    case i82547gi:
        areq = 1;
        csr32w(ctlr, Eecd, eecd|Areq);
        for(i = 0; i < 1000; i++){
            if((eecd = csr32r(ctlr, Eecd)) & Agnt)
                break;
            arch_microdelay(5);
        }
        if(!(eecd & Agnt)){
            print("igbe: not granted EEPROM access\n");
            goto release;
        }
        break;
```

```
    }
    snprint(rop, sizeof(rop), "S :%dDCc;", bits+3);

    for(addr = 0; addr < 0x40; addr++){
        /*
         * Read a word at address 'addr' from the Atmel AT93C46
         * 3-Wire Serial EEPROM or compatible. The EEPROM access is
         * controlled by 4 bits in Eecd. See the AT93C46 datasheet
         * for protocol details.
         */
        if(at93c46io(ctlr, rop, (0x06<<bits)|addr) != 0){
            print("igbe: can't set EEPROM address 0x%2.2X\n", addr);
            goto release;
        }
        data = at93c46io(ctlr, ":16COc;", 0);
        at93c46io(ctlr, "sic", 0);
        ctlr->eeprom[addr] = data;
        sum += data;
    }

release:
    if(areq)
        csr32w(ctlr, Eecd, eecd & ~Areq);
    return sum;
}
```

Uses Agnt-606 273c, Areq-605 273c, Eecd-497 271b, Eesz256-608 273c, Eeszaddr-609 273c, Spi-610 273c, at93c46io() 298, csr32r-786 279a, csr32w-787 279b, i82540em-482 271a, i82540eplp-483 271a, i82541ei-488 271a, i82541gi-489 271a, i82541gi2-490 271a, i82541pi-491 271a, i82545em-484 271a, i82545gmc-485 271a, i82546eb-493 271a, i82546gb-492 271a, i82547ei-486 271a, and i82547gi-487 271a.

⟨*function* igbedetach 301⟩≡ (307b)

```
static int
igbedetach(CtlrEtherIgbe* ctlr)
{
    int r, timeo;

    /*
     * Perform a device reset to get the chip back to the
     * power-on state, followed by an EEPROM reset to read
     * the defaults for some internal registers.
     */
    csr32w(ctlr, Imc, ~0);
    csr32w(ctlr, Rctl, 0);
    csr32w(ctlr, Tctl, 0);

    arch_delay(10);

    csr32w(ctlr, Ctrl, Devrst);
    arch_delay(1);
    for(timeo = 0; timeo < 1000; timeo++){
        if(!(csr32r(ctlr, Ctrl) & Devrst))
            break;
        arch_delay(1);
    }
    if(csr32r(ctlr, Ctrl) & Devrst)
        return -1;
    r = csr32r(ctlr, Ctrlext);
    csr32w(ctlr, Ctrlext, r|Eerst);
    arch_delay(1);
    for(timeo = 0; timeo < 1000; timeo++){
```

```
        if(!(csr32r(ctlr, Ctrlext) & Eerst))
            break;
        arch_delay(1);
    }
    if(csr32r(ctlr, Ctrlext) & Eerst)
        return -1;

    switch(ctlr->id){
    default:
        break;
    case i82540em:
    case i82540eplp:
    case i82541gi:
    case i82541gi2:
    case i82541pi:
    case i82545em:
    case i82545gmc:
    case i82547gi:
    case i82546gb:
    case i82546eb:
        r = csr32r(ctlr, Manc);
        r &= ~Arpen;
        csr32w(ctlr, Manc, r);
        break;
    }

    csr32w(ctlr, Imc, ~0);
    arch_delay(1);
    for(timeo = 0; timeo < 1000; timeo++){
        if(!csr32r(ctlr, Icr))
            break;
        arch_delay(1);
    }
    if(csr32r(ctlr, Icr))
        return -1;

    return 0;
}
```

Uses `Arpen-735` 277a, `Ctrl-494` 271b, `Ctrlext-498` 271b, `Devrst-573` 272, `Eerst-617` 274a, `Icr-503` 271b, `Imc-506` 271b, `Manc-555` 271b, `Rctl-507` 271b, `Tctl-511` 271b, `csr32r-786` 279a, `csr32w-787` 279b, `i82540em-482` 271a, `i82540eplp-483` 271a, `i82541gi-489` 271a, `i82541gi2-490` 271a, `i82541pi-491` 271a, `i82545em-484` 271a, `i82545gmc-485` 271a, `i82546eb-493` 271a, `i82546gb-492` 271a, and `i82547gi-487` 271a.

⟨*function* `igbeshutdown` 302a⟩≡ (307b)

```
static void
igbeshutdown(Ether* ether)
{
    igbedetach(ether->ctlr);
}
```

Uses `igbedetach()` 301.

⟨*function* `igbereset` 302b⟩≡ (307b)

```
static int
igbereset(CtlrEtherIgbe* ctlr)
{
    int ctrl, i, pause, r, swdpio, txcw;

    if(igbedetach(ctlr))
        return -1;
```

```
/*
 * Read the EEPROM, validate the checksum
 * then get the device back to a power-on state.
 */
if((r = at93c46r(ctlr)) != 0xBABA){
    print("igbe: bad EEPROM checksum - 0x%4.4uX\n", r);
    return -1;
}

/*
 * Snarf and set up the receive addresses.
 * There are 16 addresses. The first should be the MAC address.
 * The others are cleared and not marked valid (MS bit of Rah).
 */
if ((ctlr->id == i82546gb || ctlr->id == i82546eb) &&
    BUSFNO(ctlr->pcidev->tbdf) == 1)
    ctlr->eeprom[Ea+2] += 0x100;        /* second interface */
if(ctlr->id == i82541gi && ctlr->eeprom[Ea] == 0xFFFF)
    ctlr->eeprom[Ea] = 0xD000;
for(i = Ea; i < Eaddrlen/2; i++){
    ctlr->ra[2*i] = ctlr->eeprom[i];
    ctlr->ra[2*i+1] = ctlr->eeprom[i]>>8;
}
/* lan id seems to vary on 82543gc; don't use it */
if (ctlr->id != i82543gc) {
    r = (csr32r(ctlr, Status) & Lanid) >> 2;
    ctlr->ra[5] += r;       /* ea ctlr[1] = ea ctlr[0]+1 */
}

r = (ctlr->ra[3]<<24)|(ctlr->ra[2]<<16)|(ctlr->ra[1]<<8)|ctlr->ra[0];
csr32w(ctlr, Ral, r);
r = 0x80000000|(ctlr->ra[5]<<8)|ctlr->ra[4];
csr32w(ctlr, Rah, r);
for(i = 1; i < 16; i++){
    csr32w(ctlr, Ral+i*8, 0);
    csr32w(ctlr, Rah+i*8, 0);
}

/*
 * Clear the Multicast Table Array.
 * It's a 4096 bit vector accessed as 128 32-bit registers.
 */
memset(ctlr->mta, 0, sizeof(ctlr->mta));
for(i = 0; i < 128; i++)
    csr32w(ctlr, Mta+i*4, 0);

/*
 * Just in case the Eerst didn't load the defaults
 * (doesn't appear to fully on the 82543GC), do it manually.
 */
if (ctlr->id == i82543gc) {
    txcw = csr32r(ctlr, Txcw);
    txcw &= ~(TxcwAne|TxcwPauseMASK|TxcwFd);
    ctrl = csr32r(ctlr, Ctrl);
    ctrl &= ~(SwdpioloMASK|Frcspd|Ilos|Lrst|Fd);

    if(ctlr->eeprom[Icw1] & 0x0400){
        ctrl |= Fd;
        txcw |= TxcwFd;
    }
```

```
        if(ctlr->eeprom[Icw1] & 0x0200)
            ctrl |= Lrst;
        if(ctlr->eeprom[Icw1] & 0x0010)
            ctrl |= Ilos;
        if(ctlr->eeprom[Icw1] & 0x0800)
            ctrl |= Frcspd;
        swdpio = (ctlr->eeprom[Icw1] & 0x01E0)>>5;
        ctrl |= swdpio<<SwdpioloSHIFT;
        csr32w(ctlr, Ctrl, ctrl);

        ctrl = csr32r(ctlr, Ctrlext);
        ctrl &= ~(Ips|SwdpiohiMASK);
        swdpio = (ctlr->eeprom[Icw2] & 0x00F0)>>4;
        if(ctlr->eeprom[Icw1] & 0x1000)
            ctrl |= Ips;
        ctrl |= swdpio<<SwdpiohiSHIFT;
        csr32w(ctlr, Ctrlext, ctrl);

        if(ctlr->eeprom[Icw2] & 0x0800)
            txcw |= TxcwAne;
        pause = (ctlr->eeprom[Icw2] & 0x3000)>>12;
        txcw |= pause<<TxcwPauseSHIFT;
        switch(pause){
        default:
            ctlr->fcrtl = 0x00002000;
            ctlr->fcrth = 0x00004000;
            txcw |= TxcwAs|TxcwPs;
            break;
        case 0:
            ctlr->fcrtl = 0x00002000;
            ctlr->fcrth = 0x00004000;
            break;
        case 2:
            ctlr->fcrtl = 0;
            ctlr->fcrth = 0;
            txcw |= TxcwAs;
            break;
        }
        ctlr->txcw = txcw;
        csr32w(ctlr, Txcw, txcw);
    }


    /*
     * Flow control - values from the datasheet.
     */
    csr32w(ctlr, Fcal, 0x00C28001);
    csr32w(ctlr, Fcah, 0x00000100);
    csr32w(ctlr, Fct, 0x00008808);
    csr32w(ctlr, Fcttv, 0x00000100);

    csr32w(ctlr, Fcrtl, ctlr->fcrtl);
    csr32w(ctlr, Fcrth, ctlr->fcrth);

    if(!(csr32r(ctlr, Status) & Tbimode) && igbemii(ctlr) < 0)
        return -1;

    return 0;
}
```

Uses Ctrl-494 271b, Ctrlext-498 271b, Ea-620 274b, Eaddrlen 27d, Fcah-501 271b, Fcal-500 271b, Fcrth-516 271b, Fcrtl-515 271b, Fct-502 271b, Fcttv-508 271b, Fd-595 273b, Frcspd-567 272, Icw1-623 274b, Icw2-628 274b, Ilos-561 272, Ips-618 274a, Lanid-578 273a, Lrst-558 272, Mta-552 271b, Rah-554 271b, Ral-553 271b, Status-496 271b, SwdpiohiMASK-614 274a, SwdpiohiSHIFT-615 274a, SwdpioloMASK-571 272, SwdpioloSHIFT-572 272, Tbimode-580 273a, Txcw-509 271b, TxcwAne-669 275b, TxcwAs-664 275b, TxcwFd-659 275b, TxcwPauseMASK-661 275b, TxcwPauseSHIFT-662 275b, TxcwPs-663 275b, at93c46r() 300, csr32r-786 279a, csr32w-787 279b, i82541gi-489 271a, i82543gc-478 271a, i82546eb-493 271a, i82546gb-492 271a, igbedetach() 301, and igbemii() 296b.

⟨*function* igbepci 305⟩≡ (307b)

```
static void
igbepci(void)
{
    int cls;
    Pcidev *p;
    CtlrEtherIgbe *ctlr;
    void *mem;

    p = nil;
    while(p = pcimatch(p, 0, 0)){
        if(p->ccrb != 0x02 || p->ccru != 0)
            continue;

        switch((p->did<<16)|p->vid){
        default:
            continue;
        case i82543gc:
        case i82544ei:
        case i82544eif:
        case i82544gc:
        case i82547ei:
        case i82547gi:
        case i82540em:
        case i82540eplp:
        case i82541ei:
        case i82541gi:
        case i82541gi2:
        case i82541pi:
        case i82545em:
        case i82545gmc:
        case i82546gb:
        case i82546eb:
            break;
        }

        mem = vmap(p->mem[0].bar & ~0x0F, p->mem[0].size);
        if(mem == nil){
            print("igbe: can't map %8.8luX\n", p->mem[0].bar);
            continue;
        }
        cls = pcicfgr8(p, PciCLS);
        switch(cls){
        default:
            print("igbe: p->cls %#ux, setting to 0x10\n", p->cls);
            p->cls = 0x10;
            pcicfgw8(p, PciCLS, p->cls);
            break;
        case 0x08:
        case 0x10:
            break;
        }
```

```
        ctlr = malloc(sizeof(CtlrEtherIgbe));
        if(ctlr == nil) {
            vunmap(mem, p->mem[0].size);
            error(Enomem);
        }
        ctlr->port = p->mem[0].bar & ~0x0F;
        ctlr->pcidev = p;
        ctlr->id = (p->did<<16)|p->vid;
        ctlr->cls = cls*4;
        ctlr->nic = mem;

        if(igbereset(ctlr)){
            free(ctlr);
            vunmap(mem, p->mem[0].size);
            continue;
        }
        pcisetbme(p);

        if(igbectlrhead != nil)
            igbectlrtail->next = ctlr;
        else
            igbectlrhead = ctlr;
        igbectlrtail = ctlr;
    }
}
```

Uses `i82540em-482` 271a, `i82540eplp-483` 271a, `i82541ei-488` 271a, `i82541gi-489` 271a, `i82541gi2-490` 271a, `i82541pi-491` 271a, `i82543gc-478` 271a, `i82544ei-479` 271a, `i82544eif-480` 271a, `i82544gc-481` 271a, `i82545em-484` 271a, `i82545gmc-485` 271a, `i82546eb-493` 271a, `i82546gb-492` 271a, `i82547ei-486` 271a, `i82547gi-487` 271a, `igbectlrhead-788` 279c, `igbectlrtail-789` 279d, and `igbereset()` 302b.

⟨*function* `igbepnp` 306⟩≡ (307b)

```
static int
igbepnp(Ether* edev)
{
    CtlrEtherIgbe *ctlr;

    if(igbectlrhead == nil)
        igbepci();

    /*
     * Any adapter matches if no edev->port is supplied,
     * otherwise the ports must match.
     */
    for(ctlr = igbectlrhead; ctlr != nil; ctlr = ctlr->next){
        if(ctlr->active)
            continue;
        if(edev->port == 0 || edev->port == ctlr->port){
            ctlr->active = 1;
            break;
        }
    }
    if(ctlr == nil)
        return -1;

    edev->ctlr = ctlr;
    edev->port = ctlr->port;
    edev->irq = ctlr->pcidev->intl;
    edev->tbdf = ctlr->pcidev->tbdf;
    edev->mbps = 1000;
    memmove(edev->ea, ctlr->ra, Eaddrlen);
```

```
    /*
     * Linkage to the generic ethernet driver.
     */
    edev->attach = igbeattach;
    edev->transmit = igbetransmit;
    edev->interrupt = igbeinterrupt;
    edev->ifstat = igbeifstat;
    edev->ctl = igbectl;

    edev->arg = edev;
    edev->promiscuous = igbepromiscuous;
    edev->shutdown = igbeshutdown;
    edev->multicast = igbemulticast;

    return 0;
}
```

Uses Eaddrlen 27d, igbeattach() 292, igbectl() 282c, igbectlrhead-788 279c, igbeifstat() 281, igbeinterrupt() 293, igbemulticast() 283b, igbepci() 305, igbepromiscuous() 283a, igbeshutdown() 302a, and igbetransmit() 288.

⟨*function* etherigbelink 307a⟩≡ (307b)

```
void
etherigbelink(void)
{
    addethercard("i82543", igbepnp);
    addethercard("igbe", igbepnp);
}
```

Uses igbepnp() 306.

⟨kernel/network/386/etherigbe.c 307b⟩≡

```
/*
 * Intel 8254[340]NN Gigabit Ethernet PCI Controllers
 * as found on the Intel PRO/1000 series of adapters:
 *  82543GC Intel PRO/1000 T
 *  82544EI Intel PRO/1000 XT
 *  82540EM Intel PRO/1000 MT
 *  82541[GP]I
 *  82547GI
 *  82546GB
 *  82546EB
 * To Do:
 *  finish autonegotiation code;
 *  integrate fiber stuff back in (this ONLY handles
 *  the CAT5 cards at the moment);
 *  add tuning control via ctl file;
 *  this driver is little-endian specific.
 */
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"

#include "io.h"
#include "../port/error.h"
#include "../port/netif.h"
#include "../port/etherif.h"

#include "ethermii.h"
```

```
typedef struct Td Td;
typedef struct CtlrEtherIgbe CtlrEtherIgbe;


⟨enum _anon_ (kernel/network/386/etherigbe.c) 271a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)2 271b⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)3 272⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)4 273a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)5 273b⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)6 273c⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)7 274a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)8 274b⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)9 274c⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)10 274d⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)11 275a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)12 275b⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)13 275c⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)14 275d⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)15 276a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)16 276b⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)17 276c⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)18 277a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)19 277b⟩

⟨struct Rd 277c⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)20 277d⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)21 277e⟩

⟨struct Td 277f⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)22 278a⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)23 278b⟩

⟨enum _anon_ (kernel/network/386/etherigbe.c)24 278c⟩

⟨struct CtlrEtherIgbe 278d⟩

⟨macro csr32r 279a⟩
⟨macro csr32w 279b⟩
```

⟨*global* igbectlrhead 279c⟩
⟨*global* igbectlrtail 279d⟩

⟨*global* igberblock 280a⟩
⟨*global* igberbpool 280b⟩

⟨*global* statistics 280c⟩

⟨*function* igbeifstat 281⟩

⟨*enum* _anon_ (kernel/network/386/etherigbe.c)25 282a⟩

⟨*global* igbectlmsg 282b⟩

⟨*function* igbectl 282c⟩

⟨*function* igbepromiscuous 283a⟩

⟨*function* igbemulticast 283b⟩

⟨*function* igberballoc 284a⟩

⟨*function* igberbfree 284b⟩

⟨*function* igbeim 284c⟩

⟨*function* igbelim 285a⟩

⟨*function* igbelproc 285b⟩

⟨*function* igbetxinit 286⟩

⟨*function* igbetransmit 288⟩

⟨*function* igbereplenish 289a⟩

⟨*function* igberxinit 289b⟩

⟨*function* igberim 290a⟩

⟨*function* igberproc 290b⟩

⟨*function* igbeattach 292⟩

⟨*function* igbeinterrupt 293⟩

⟨*function* i82543mdior 294a⟩

⟨*function* i82543mdiow 294b⟩

⟨*function* i82543miimir 295a⟩

⟨*function* i82543miimiw 295b⟩

⟨*function* igbemiimir 295c⟩

⟨*function* igbemiimiw 296a⟩

⟨*function* igbemii 296b⟩

⟨*function* `at93c46io` 298⟩

⟨*function* `at93c46r` 300⟩

⟨*function* `igbedetach` 301⟩

⟨*function* `igbeshutdown` 302a⟩

⟨*function* `igbereset` 302b⟩

⟨*function* `igbepci` 305⟩

⟨*function* `igbepnp` 306⟩

⟨*function* `etherigbelink` 307a⟩

Uses `CtlrEtherIgbe` 278d and `Td` 277f.

I.6.7 `kernel/network/386/ethermii.c`

⟨*function* `mii` 310⟩≡ (314a)

```
int
mii(Mii* mii, int mask)
{
    MiiPhy *miiphy;
    int bit, oui, phyno, r, rmask;

    /*
     * Probe through mii for PHYs in mask;
     * return the mask of those found in the current probe.
     * If the PHY has not already been probed, update
     * the Mii information.
     */
    rmask = 0;
    for(phyno = 0; phyno < NMiiPhy; phyno++){
        bit = 1<<phyno;
        if(!(mask & bit))
            continue;
        if(mii->mask & bit){
            rmask |= bit;
            continue;
        }
        if(mii->mir(mii, phyno, Bmsr) == -1)
            continue;
        r = mii->mir(mii, phyno, Phyidr1);
        oui = (r & 0x3FFF)<<6;
        r = mii->mir(mii, phyno, Phyidr2);
        oui |= r>>10;
        if(oui == 0xFFFFF || oui == 0)
            continue;

        if((miiphy = malloc(sizeof(MiiPhy))) == nil)
            continue;

        miiphy->mii = mii;
        miiphy->oui = oui;
        miiphy->phyno = phyno;
```

```
        miiphy->anar = ~0;
        miiphy->fc = ~0;
        miiphy->mscr = ~0;

        mii->phy[phyno] = miiphy;
        if(mii->curphy == nil)
            mii->curphy = miiphy;
        mii->mask |= bit;
        mii->nphy++;

        rmask |= bit;
    }
    return rmask;
}
```
Uses Bmsr 314b, NMiiPhy 314b, Phyidr1 314b, and Phyidr2 314b.

⟨*function* miimir 311a⟩≡ (314a)
```
int
miimir(Mii* mii, int r)
{
    if(mii == nil || mii->ctlr == nil || mii->curphy == nil)
        return -1;
    return mii->mir(mii, mii->curphy->phyno, r);
}
```

⟨*function* miimiw 311b⟩≡ (314a)
```
int
miimiw(Mii* mii, int r, int data)
{
    if(mii == nil || mii->ctlr == nil || mii->curphy == nil)
        return -1;
    return mii->miw(mii, mii->curphy->phyno, r, data);
}
```

⟨*function* miireset 311c⟩≡ (314a)
```
int
miireset(Mii* mii)
{
    int bmcr;

    if(mii == nil || mii->ctlr == nil || mii->curphy == nil)
        return -1;
    bmcr = mii->mir(mii, mii->curphy->phyno, Bmcr);
    bmcr |= BmcrR;
    mii->miw(mii, mii->curphy->phyno, Bmcr, bmcr);
    arch_microdelay(1);

    return 0;
}
```
Uses Bmcr 314b and BmcrR 315a.

⟨*function* miiane 311d⟩≡ (314a)
```
int
miiane(Mii* mii, int a, int p, int e)
{
    int anar, bmsr, mscr, r, phyno;

    if(mii == nil || mii->ctlr == nil || mii->curphy == nil)
        return -1;
    phyno = mii->curphy->phyno;
```

```
    bmsr = mii->mir(mii, phyno, Bmsr);
    if(!(bmsr & BmsrAna))
        return -1;

    if(a != ~0)
        anar = (AnaTXFD|AnaTXHD|Ana10FD|Ana10HD) & a;
    else if(mii->curphy->anar != ~0)
        anar = mii->curphy->anar;
    else{
        anar = mii->mir(mii, phyno, Anar);
        anar &= ~(AnaAP|AnaP|AnaT4|AnaTXFD|AnaTXHD|Ana10FD|Ana10HD);
        if(bmsr & Bmsr10THD)
            anar |= Ana10HD;
        if(bmsr & Bmsr10TFD)
            anar |= Ana10FD;
        if(bmsr & Bmsr100TXHD)
            anar |= AnaTXHD;
        if(bmsr & Bmsr100TXFD)
            anar |= AnaTXFD;
    }
    mii->curphy->anar = anar;

    if(p != ~0)
        anar |= (AnaAP|AnaP) & p;
    else if(mii->curphy->fc != ~0)
        anar |= mii->curphy->fc;
    mii->curphy->fc = (AnaAP|AnaP) & anar;

    if(bmsr & BmsrEs){
        mscr = mii->mir(mii, phyno, Mscr);
        mscr &= ~(Mscr1000TFD|Mscr1000THD);
        if(e != ~0)
            mscr |= (Mscr1000TFD|Mscr1000THD) & e;
        else if(mii->curphy->mscr != ~0)
            mscr = mii->curphy->mscr;
        else{
            r = mii->mir(mii, phyno, Esr);
            if(r & Esr1000THD)
                mscr |= Mscr1000THD;
            if(r & Esr1000TFD)
                mscr |= Mscr1000TFD;
        }
        mii->curphy->mscr = mscr;
        mii->miw(mii, phyno, Mscr, mscr);
    }
    mii->miw(mii, phyno, Anar, anar);

    r = mii->mir(mii, phyno, Bmcr);
    if(!(r & BmcrR)){
        r |= BmcrAne|BmcrRan;
        mii->miw(mii, phyno, Bmcr, r);
    }

    return 0;
}
```

Uses Ana10FD 315c, Ana10HD 315c, AnaAP 315c, AnaP 315c, AnaT4 315c, AnaTXFD 315c, AnaTXHD 315c, Anar 314b, Bmcr 314b, BmcrAne 315a, BmcrR 315a, BmcrRan 315a, Bmsr 314b, Bmsr100TXFD 315b, Bmsr100TXHD 315b, Bmsr10TFD 315b, Bmsr10THD 315b, BmsrAna 315b, BmsrEs 315b, Esr 314b, Esr1000TFD 316b, Esr1000THD 316b, Mscr 314b, Mscr1000TFD 315d,

and Mscr1000THD 315d.

⟨*function* miistatus 313⟩≡ (314a)

```
int
miistatus(Mii* mii)
{
    MiiPhy *phy;
    int anlpar, bmsr, p, r, phyno;

    if(mii == nil || mii->ctlr == nil || mii->curphy == nil)
        return -1;
    phy = mii->curphy;
    phyno = phy->phyno;

    /*
     * Check Auto-Negotiation is complete and link is up.
     * (Read status twice as the Ls bit is sticky).
     */
    bmsr = mii->mir(mii, phyno, Bmsr);
    if(!(bmsr & (BmsrAnc|BmsrAna))) {
        // print("miistatus: auto-neg incomplete\n");
        return -1;
    }

    bmsr = mii->mir(mii, phyno, Bmsr);
    if(!(bmsr & BmsrLs)){
        // print("miistatus: link down\n");
        phy->link = 0;
        return -1;
    }

    phy->speed = phy->fd = phy->rfc = phy->tfc = 0;
    if(phy->mscr){
        r = mii->mir(mii, phyno, Mssr);
        if((phy->mscr & Mscr1000TFD) && (r & Mssr1000TFD)){
            phy->speed = 1000;
            phy->fd = 1;
        }
        else if((phy->mscr & Mscr1000THD) && (r & Mssr1000THD))
            phy->speed = 1000;
    }

    anlpar = mii->mir(mii, phyno, Anlpar);
    if(phy->speed == 0){
        r = phy->anar & anlpar;
        if(r & AnaTXFD){
            phy->speed = 100;
            phy->fd = 1;
        }
        else if(r & AnaTXHD)
            phy->speed = 100;
        else if(r & Ana10FD){
            phy->speed = 10;
            phy->fd = 1;
        }
        else if(r & Ana10HD)
            phy->speed = 10;
    }
    if(phy->speed == 0) {
        // print("miistatus: phy speed 0\n");
```

```
        return -1;
    }

    if(phy->fd){
        p = phy->fc;
        r = anlpar & (AnaAP|AnaP);
        if(p == AnaAP && r == (AnaAP|AnaP))
            phy->tfc = 1;
        else if(p == (AnaAP|AnaP) && r == AnaAP)
            phy->rfc = 1;
        else if((p & AnaP) && (r & AnaP))
            phy->rfc = phy->tfc = 1;
    }

    phy->link = 1;

    return 0;
}
```

Uses `Ana10FD` 315c, `Ana10HD` 315c, `AnaAP` 315c, `AnaP` 315c, `AnaTXFD` 315c, `AnaTXHD` 315c, `Anlpar` 314b, `Bmsr` 314b, `BmsrAna` 315b, `BmsrAnc` 315b, `BmsrLs` 315b, `Mscr1000TFD` 315d, `Mscr1000THD` 315d, `Mssr` 314b, `Mssr1000TFD` 316a, and `Mssr1000THD` 316a.

⟨`kernel/network/386/ethermii.c` 314a⟩≡

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"
#include "../port/error.h"
#include "../port/netif.h"
#include "../port/etherif.h"

#include "ethermii.h"

⟨function mii 310⟩

⟨function miimir 311a⟩

⟨function miimiw 311b⟩

⟨function miireset 311c⟩

⟨function miiane 311d⟩

⟨function miistatus 313⟩
```

I.6.8 kernel/network/386/ethermii.h

⟨*enum* `_anon_ (kernel/network/386/ethermii.h)` 314b⟩≡ (316e)

```
enum {          /* registers */
  Bmcr    = 0x00,   /* Basic Mode Control */
  Bmsr    = 0x01,   /* Basic Mode Status */
  Phyidr1   = 0x02,   /* PHY Identifier #1 */
  Phyidr2   = 0x03,   /* PHY Identifier #2 */
  Anar    = 0x04,   /* Auto-Negotiation Advertisement */
  Anlpar    = 0x05,   /* AN Link Partner Ability */
  Aner    = 0x06,   /* AN Expansion */
  Annptr    = 0x07,   /* AN Next Page TX */
  Annprr    = 0x08,   /* AN Next Page RX */
```

```
    Mscr    = 0x09,   /* MASTER-SLAVE Control */
    Mssr    = 0x0A,   /* MASTER-SLAVE Status */
    Esr   = 0x0F,   /* Extended Status */

    NMiiPhyr  = 32,
    NMiiPhy   = 32,
  };
```

⟨*enum* _anon_ (kernel/network/386/ethermii.h)2 315a⟩≡ (316e)

```
  enum {          /* Bmcr */
    BmcrSs1   = 0x0040, /* Speed Select[1] */
    BmcrCte   = 0x0080, /* Collision Test Enable */
    BmcrDm    = 0x0100, /* Duplex Mode */
    BmcrRan   = 0x0200, /* Restart Auto-Negotiation */
    BmcrI   = 0x0400, /* Isolate */
    BmcrPd    = 0x0800, /* Power Down */
    BmcrAne   = 0x1000, /* Auto-Negotiation Enable */
    BmcrSs0   = 0x2000, /* Speed Select[0] */
    BmcrLe    = 0x4000, /* Loopback Enable */
    BmcrR   = 0x8000, /* Reset */
  };
```

⟨*enum* _anon_ (kernel/network/386/ethermii.h)3 315b⟩≡ (316e)

```
  enum {          /* Bmsr */
    BmsrEc    = 0x0001, /* Extended Capability */
    BmsrJd    = 0x0002, /* Jabber Detect */
    BmsrLs    = 0x0004, /* Link Status */
    BmsrAna   = 0x0008, /* Auto-Negotiation Ability */
    BmsrRf    = 0x0010, /* Remote Fault */
    BmsrAnc   = 0x0020, /* Auto-Negotiation Complete */
    BmsrPs    = 0x0040, /* Preamble Suppression Capable */
    BmsrEs    = 0x0100, /* Extended Status */
    Bmsr100T2HD = 0x0200, /* 100BASE-T2 HD Capable */
    Bmsr100T2FD = 0x0400, /* 100BASE-T2 FD Capable */
    Bmsr10THD = 0x0800, /* 10BASE-T HD Capable */
    Bmsr10TFD = 0x1000, /* 10BASE-T FD Capable */
    Bmsr100TXHD = 0x2000, /* 100BASE-TX HD Capable */
    Bmsr100TXFD = 0x4000, /* 100BASE-TX FD Capable */
    Bmsr100T4 = 0x8000, /* 100BASE-T4 Capable */
  };
```

⟨*enum* _anon_ (kernel/network/386/ethermii.h)4 315c⟩≡ (316e)

```
  enum {          /* Anar/Anlpar */
    Ana10HD   = 0x0020, /* Advertise 10BASE-T */
    Ana10FD   = 0x0040, /* Advertise 10BASE-T FD */
    AnaTXHD   = 0x0080, /* Advertise 100BASE-TX */
    AnaTXFD   = 0x0100, /* Advertise 100BASE-TX FD */
    AnaT4   = 0x0200, /* Advertise 100BASE-T4 */
    AnaP    = 0x0400, /* Pause */
    AnaAP   = 0x0800, /* Asymmetrical Pause */
    AnaRf   = 0x2000, /* Remote Fault */
    AnaAck    = 0x4000, /* Acknowledge */
    AnaNp   = 0x8000, /* Next Page Indication */
  };
```

⟨*enum* _anon_ (kernel/network/386/ethermii.h)5 315d⟩≡ (316e)

```
  enum {          /* Mscr */
    Mscr1000THD = 0x0100, /* Advertise 1000BASE-T HD */
    Mscr1000TFD = 0x0200, /* Advertise 1000BASE-T FD */
  };
```

⟨*enum* _anon_ (kernel/network/386/ethermii.h)6 316a⟩≡ (316e)

```
enum {          /* Mssr */
  Mssr1000THD = 0x0400, /* Link Partner 1000BASE-T HD able */
  Mssr1000TFD = 0x0800, /* Link Partner 1000BASE-T FD able */
};
```

⟨*enum* _anon_ (kernel/network/386/ethermii.h)7 316b⟩≡ (316e)

```
enum {          /* Esr */
  Esr1000THD  = 0x1000, /* 1000BASE-T HD Capable */
  Esr1000TFD  = 0x2000, /* 1000BASE-T FD Capable */
  Esr1000XHD  = 0x4000, /* 1000BASE-X HD Capable */
  Esr1000XFD  = 0x8000, /* 1000BASE-X FD Capable */
};
```

⟨*struct* Mii 316c⟩≡ (316e)

```
struct Mii {
  Lock;
  int nphy;
  int mask;
  MiiPhy* phy[NMiiPhy];
  MiiPhy* curphy;

  void* ctlr;
  int (*mir)(Mii*, int, int);
  int (*miw)(Mii*, int, int, int);
};
```

Uses NMiiPhy 314b.

⟨*struct* MiiPhy 316d⟩≡ (316e)

```
struct MiiPhy {
  Mii*  mii;
  int oui;
  int phyno;

  int anar;
  int fc;
  int mscr;

  int link;
  int speed;
  int fd;
  int rfc;
  int tfc;
};
```

⟨kernel/network/386/ethermii.h 316e⟩≡

```
typedef struct Mii Mii;
typedef struct MiiPhy MiiPhy;

⟨enum _anon_ (kernel/network/386/ethermii.h) 314b⟩

⟨enum _anon_ (kernel/network/386/ethermii.h)2 315a⟩

⟨enum _anon_ (kernel/network/386/ethermii.h)3 315b⟩

⟨enum _anon_ (kernel/network/386/ethermii.h)4 315c⟩

⟨enum _anon_ (kernel/network/386/ethermii.h)5 315d⟩

⟨enum _anon_ (kernel/network/386/ethermii.h)6 316a⟩
```

```
⟨enum _anon_ (kernel/network/386/ethermii.h)7 316b⟩

⟨struct Mii 316c⟩

⟨struct MiiPhy 316d⟩

extern int mii(Mii*, int);
extern int miiane(Mii*, int, int, int);
extern int miimir(Mii*, int);
extern int miimiw(Mii*, int, int);
extern int miireset(Mii*);
extern int miistatus(Mii*);
```
Uses `Mii` 316c and `MiiPhy` 316d.

I.7 kernel/network/arm/

I.7.1 network/arm/devether.c

⟨*global* `etherxx`*(arm)* 317a⟩≡ (326c)
```
static Ether *etherxx[MaxEther];
```
Uses `MaxEther` 28d.

⟨*function* `etherattach`*(arm)* 317b⟩≡ (326c)
```
Chan*
etherattach(char* spec)
{
    int ctlrno;
    char *p;
    Chan *chan;

    ctlrno = 0;
    if(spec && *spec){
        ctlrno = strtoul(spec, &p, 0);
        if((ctlrno == 0 && p == spec) || *p != 0)
            error(Ebadarg);
        if(ctlrno < 0 || ctlrno >= MaxEther)
            error(Ebadarg);
    }
    if(etherxx[ctlrno] == 0)
        error(Enodev);

    chan = devattach('l', spec);
    if(waserror()){
        chanfree(chan);
        nexterror();
    }
    chan->dev = ctlrno;
    if(etherxx[ctlrno]->attach)
        etherxx[ctlrno]->attach(etherxx[ctlrno]);
    poperror();
    return chan;
}
```

⟨*function* `etherwalk`*(arm)* 317c⟩≡ (326c)
```
static Walkqid*
etherwalk(Chan* chan, Chan* nchan, char** name, int nname)
{
```

```
	return netifwalk(etherxx[chan->dev], chan, nchan, name, nname);
}
```
Uses etherxx-798 317a and netifwalk() 244a.

⟨*function* etherstat*(arm)* 318a⟩≡ (326c)
```
static int
etherstat(Chan* chan, uchar* dp, int n)
{
	return netifstat(etherxx[chan->dev], chan, dp, n);
}
```
Uses etherxx-798 317a and netifstat() 248a.

⟨*function* etheropen*(arm)* 318b⟩≡ (326c)
```
static Chan*
etheropen(Chan* chan, int omode)
{
	return netifopen(etherxx[chan->dev], chan, omode);
}
```
Uses etherxx-798 317a and netifopen() 244b.

⟨*function* ethercreate*(arm)* 318c⟩≡ (326c)
```
static void
ethercreate(Chan*, char*, int, ulong)
{
}
```

⟨*function* etherclose*(arm)* 318d⟩≡ (326c)
```
static void
etherclose(Chan* chan)
{
	netifclose(etherxx[chan->dev], chan);
}
```
Uses etherxx-798 317a and netifclose() 248b.

⟨*function* etherread*(arm)* 318e⟩≡ (326c)
```
static long
etherread(Chan* chan, void* buf, long n, vlong off)
{
	Ether *ether;
	ulong offset = off;

	ether = etherxx[chan->dev];
	if((chan->qid.type & QTDIR) == 0 && ether->ifstat){
		/*
		 * With some controllers it is necessary to reach
		 * into the chip to extract statistics.
		 */
		if(NETTYPE(chan->qid.path) == Nifstatqid)
			return ether->ifstat(ether, buf, n, offset);
		else if(NETTYPE(chan->qid.path) == Nstatqid)
			ether->ifstat(ether, buf, 0, offset);
	}

	return netifread(ether, chan, buf, n, offset);
}
```
Uses NETTYPE 240b, Nifstatqid 240a, Nstatqid 240a, etherxx-798 317a, and netifread() 245.

⟨*function* etherbread*(arm)* 319a⟩≡ (326c)
```
static Block*
etherbread(Chan* chan, long n, ulong offset)
{
    return netifbread(etherxx[chan->dev], chan, n, offset);
}
```
Uses etherxx-798 317a and netifbread() 246a.

⟨*function* etherwstat*(arm)* 319b⟩≡ (326c)
```
static int
etherwstat(Chan* chan, uchar* dp, int n)
{
    return netifwstat(etherxx[chan->dev], chan, dp, n);
}
```
Uses etherxx-798 317a and netifwstat() 247.

⟨*function* etherrtrace*(arm)* 319c⟩≡ (326c)
```
static void
etherrtrace(Netfile* f, Etherpkt* pkt, int len)
{
    int i, n;
    Block *bp;

    if(qwindow(f->in) <= 0)
        return;
    if(len > 58)
        n = 58;
    else
        n = len;
    bp = iallocb(64);
    if(bp == nil)
        return;
    memmove(bp->wp, pkt->d, n);
    i = TK2MS(CPUS(0)->ticks);
    bp->wp[58] = len>>8;
    bp->wp[59] = len;
    bp->wp[60] = i>>24;
    bp->wp[61] = i>>16;
    bp->wp[62] = i>>8;
    bp->wp[63] = i;
    bp->wp += 64;
    qpass(f->in, bp);
}
```

⟨*function* etheriq*(arm)* 319d⟩≡ (326c)
```
Block*
etheriq(Ether* ether, Block* bp, int fromwire)
{
    Etherpkt *pkt;
    ushort type;
    int len, multi, tome, fromme;
    Netfile **ep, *f, **fp, *fx;
    Block *xbp;

    ether->inpackets++;

    pkt = (Etherpkt*)bp->rp;
    len = BLEN(bp);
    type = (pkt->type[0]<<8)|pkt->type[1];
    fx = 0;
```

```
    ep = &ether->f[Ntypes];

    multi = pkt->d[0] & 1;
    /* check for valid multicast addresses */
    if(multi && memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) != 0 &&
        ether->prom == 0){
        if(!activemulti(ether, pkt->d, sizeof(pkt->d))){
            if(fromwire){
                freeb(bp);
                bp = 0;
            }
            return bp;
        }
    }
    /* is it for me? */
    tome = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
    fromme = memcmp(pkt->s, ether->ea, sizeof(pkt->s)) == 0;

    /*
     * Multiplex the packet to all the connections which want it.
     * If the packet is not to be used subsequently (fromwire != 0),
     * attempt to simply pass it into one of the connections, thereby
     * saving a copy of the data (usual case hopefully).
     */
    for(fp = ether->f; fp < ep; fp++){
        if((f = *fp) != nil && (f->type == type || f->type < 0) &&
            (tome || multi || f->prom)){
            /* Don't want to hear bridged packets */
            if(f->bridge && !fromwire && !fromme)
                continue;
            if(!f->headersonly){
                if(fromwire && fx == 0)
                    fx = f;
                else if(xbp = iallocb(len)){
                    memmove(xbp->wp, pkt, len);
                    xbp->wp += len;
                    if(qpass(f->in, xbp) < 0)
                        ether->soverflows++;
                }
                else
                    ether->soverflows++;
            }
            else
                etherrtrace(f, pkt, len);
        }
    }

    if(fx){
        if(qpass(fx->in, bp) < 0)
            ether->soverflows++;
        return 0;
    }
    if(fromwire){
        freeb(bp);
        return 0;
    }
    return bp;
}
```

Uses `Ntypes` 255a and `activemulti()` 252c.

⟨*function* etheroq*(arm)* 321a⟩≡ (326c)
```
static int
etheroq(Ether* ether, Block* bp)
{
    int len, loopback, s;
    Etherpkt *pkt;

    ether->outpackets++;

    /*
     * Check if the packet has to be placed back onto the input queue,
     * i.e. if it's a loopback or broadcast packet or the interface is
     * in promiscuous mode.
     * If it's a loopback packet indicate to etheriq that the data isn't
     * needed and return, etheriq will pass-on or free the block.
     * To enable bridging to work, only packets that were originated
     * by this interface are fed back.
     */
    pkt = (Etherpkt*)bp->rp;
    len = BLEN(bp);
    loopback = memcmp(pkt->d, ether->ea, sizeof(pkt->d)) == 0;
    if(loopback || memcmp(pkt->d, ether->bcast, sizeof(pkt->d)) == 0 || ether->prom){
        s = arch_splhi();
        etheriq(ether, bp, 0);
        arch_splx(s);
    }

    if(!loopback){
        qbwrite(ether->oq, bp);
        if(ether->transmit != nil)
            ether->transmit(ether);
    } else
        freeb(bp);

    return len;
}
```

⟨*function* etherwrite*(arm)* 321b⟩≡ (326c)
```
static long
etherwrite(Chan* chan, void* buf, long n, vlong)
{
    Ether *ether;
    Block *bp;
    int nn, onoff;
    Cmdbuf *cb;

    ether = etherxx[chan->dev];
    if(NETTYPE(chan->qid.path) != Ndataqid) {
        nn = netifwrite(ether, chan, buf, n);
        if(nn >= 0)
            return nn;
        cb = parsecmd(buf, n);
        if(cb->f[0] && strcmp(cb->f[0], "nonblocking") == 0){
            if(cb->nf <= 1)
                onoff = 1;
            else
                onoff = atoi(cb->f[1]);
            qnoblock(ether->oq, onoff);
            free(cb);
            return n;
```

```
        }
        free(cb);
        if(ether->ctl!=nil)
            return ether->ctl(ether,buf,n);

        error(Ebadctl);
    }

    if(n > ether->maxmtu)
        error(Etoobig);
    if(n < ether->minmtu)
        error(Etoosmall);

    bp = allocb(n);
    if(waserror()){
        freeb(bp);
        nexterror();
    }
    memmove(bp->rp, buf, n);
    memmove(bp->rp+Eaddrlen, ether->ea, Eaddrlen);
    poperror();
    bp->wp += n;

    return etheroq(ether, bp);
}
```

Uses `Eaddrlen` 27d, `NETTYPE` 240b, `Ndataqid` 240a, `etherxx-798` 317a, `netifwrite()` 246c, and `parsecmd()`.

⟨*function* `etherbwrite` *(arm)* 322⟩≡ (326c)

```
static long
etherbwrite(Chan* chan, Block* bp, ulong)
{
    Ether *ether;
    long n;

    n = BLEN(bp);
    if(NETTYPE(chan->qid.path) != Ndataqid){
        if(waserror()) {
            freeb(bp);
            nexterror();
        }
        n = etherwrite(chan, bp->rp, n, 0);
        poperror();
        freeb(bp);
        return n;
    }
    ether = etherxx[chan->dev];

    if(n > ether->maxmtu){
        freeb(bp);
        error(Etoobig);
    }
    if(n < ether->minmtu){
        freeb(bp);
        error(Etoosmall);
    }

    return etheroq(ether, bp);
}
```

Uses `NETTYPE` 240b, `Ndataqid` 240a, and `etherxx-798` 317a.

⟨*global* `cards` *(arm)* 323a⟩≡ (326c)

```
static struct {
    char* type;
    int (*reset)(Ether*);
} cards[MaxEther+1];
```

Uses `MaxEther` 28d and `__anon_struct_90` 323a.

⟨*function* `addethercard` *(arm)* 323b⟩≡ (326c)

```
void
addethercard(char* t, int (*r)(Ether*))
{
    static int ncard;

    if(ncard == MaxEther)
        panic("too many ether cards");
    cards[ncard].type = t;
    cards[ncard].reset = r;
    ncard++;
}
```

Uses `MaxEther` 28d and `cards-799` 323a.

⟨*function* `parseether` *(arm)* 323c⟩≡ (326c)

```
int
parseether(uchar *to, char *from)
{
    char nip[4];
    char *p;
    int i;

    p = from;
    for(i = 0; i < Eaddrlen; i++){
        if(*p == 0)
            return -1;
        nip[0] = *p++;
        if(*p == 0)
            return -1;
        nip[1] = *p++;
        nip[2] = 0;
        to[i] = strtoul(nip, 0, 16);
        if(*p == ':')
            p++;
    }
    return 0;
}
```

Uses `Eaddrlen` 27d.

⟨*function* `etherreset` *(arm)* 323d⟩≡ (326c)

```
static void
etherreset(void)
{
    Ether *ether;
    int i, n, ctlrno;
    char name[KNAMELEN], buf[128];

    for(ether = 0, ctlrno = 0; ctlrno < MaxEther; ctlrno++){
        if(ether == 0)
            ether = malloc(sizeof(Ether));
        memset(ether, 0, sizeof(Ether));
        ether->ctlrno = ctlrno;
```

```
        ether->mbps = 10;
        ether->minmtu = ETHERMINTU;
        ether->maxmtu = ETHERMAXTU;

        if(archether(ctlrno, ether) <= 0)
            continue;

        if(arch_isaconfig("ether", ctlrno, ether) == 0){
//   free(ether);
//   return nil;
            continue;
        }
        for(n = 0; cards[n].type; n++){
            if(cistrcmp(cards[n].type, ether->type))
                continue;
            for(i = 0; i < ether->nopt; i++)
                if(cistrncmp(ether->opt[i], "ea=", 3) == 0){
                    if(parseether(ether->ea,
                        &ether->opt[i][3]) == -1)
                        memset(ether->ea, 0, Eaddrlen);
                } else if(cistrcmp(ether->opt[i],
                    "100BASE-TXFD") == 0)
                    ether->mbps = 100;
            if(cards[n].reset(ether))
                break;
            snprint(name, sizeof(name), "ether%d", ctlrno);

            if(ether->interrupt != nil && ether->irq >= 0)
                arch_intrenable(ether->irq, ether->interrupt,
                    ether, 0, name);

            i = snprint(buf, sizeof buf,
                "#l%d: %s: %dMbps port %#lux irq %d",
                ctlrno, ether->type, ether->mbps, ether->port,
                ether->irq);
            if(ether->mem)
                i += snprint(buf+i, sizeof buf - i,
                    " addr %#lux", PADDR(ether->mem));
            if(ether->size)
                i += snprint(buf+i, sizeof buf - i,
                    " size %#luX", ether->size);
            i += snprint(buf+i, sizeof buf - i,
                ": %2.2ux%2.2ux%2.2ux%2.2ux%2.2ux%2.2ux",
                ether->ea[0], ether->ea[1], ether->ea[2],
                ether->ea[3], ether->ea[4], ether->ea[5]);
            snprint(buf+i, sizeof buf - i, "\n");
            iprint("%s", buf);  /* it may be too early for print */

            if(ether->mbps >= 1000)
                netifinit(ether, name, Ntypes, 4*1024*1024);
            else if(ether->mbps >= 100)
                netifinit(ether, name, Ntypes, 1024*1024);
            else
                netifinit(ether, name, Ntypes, 65*1024);
            if(ether->oq == 0)
                ether->oq = qopen(ether->limit, Qmsg, 0, 0);
            if(ether->oq == 0)
                panic("etherreset %s", name);
            ether->alen = Eaddrlen;
            memmove(ether->addr, ether->ea, Eaddrlen);
```

```
            memset(ether->bcast, 0xFF, Eaddrlen);

            etherxx[ctlrno] = ether;
            ether = 0;
            break;
        }
    }
    if(ether)
        free(ether);
}
```

Uses ETHERMAXTU 240f, ETHERMINTU 240f, Eaddrlen 27d, MaxEther 28d, Ntypes 255a, cards-799 323a, etherxx-798 317a, and netifinit() 241b.

⟨*function* ethershutdown*(arm)* 325a⟩≡ (326c)

```
static void
ethershutdown(void)
{
    Ether *ether;
    int i;

    for(i = 0; i < MaxEther; i++){
        ether = etherxx[i];
        if(ether == nil)
            continue;
        if(ether->shutdown == nil) {
            print("#l%d: no shutdown function\n", i);
            continue;
        }
        (*ether->shutdown)(ether);
    }
}
```

Uses MaxEther 28d and etherxx-798 317a.

⟨*constant* POLY*(arm)* 325b⟩≡ (326c)

```
#define POLY 0xedb88320
```

⟨*function* ethercrc*(arm)* 325c⟩≡ (326c)

```
/* really slow 32 bit crc for ethers */
ulong
ethercrc(uchar *p, int len)
{
    int i, j;
    ulong crc, b;

    crc = 0xffffffff;
    for(i = 0; i < len; i++){
        b = *p++;
        for(j = 0; j < 8; j++){
            crc = (crc>>1) ^ (((crc^b) & 1) ? POLY : 0);
            b >>= 1;
        }
    }
    return crc;
}
```

Uses POLY-800 325b.

⟨*function* dumpoq*(arm)* 325d⟩≡ (326c)

```
void
dumpoq(Queue *oq)
```

```
{
    if (oq == nil)
        print("no outq! ");
    else if (qisclosed(oq))
        print("outq closed ");
    else if (qfull(oq))
        print("outq full ");
    else
        print("outq %d ", qlen(oq));
}
```

⟨*function* dumpnetif *(arm)* 326a⟩≡ (326c)

```
void
dumpnetif(Netif *netif)
{
    print("netif %s ", netif->name);
    print("limit %d mbps %d link %d ",
        netif->limit, netif->mbps, netif->link);
    print("inpkts %lld outpkts %lld errs %d\n",
        netif->inpackets, netif->outpackets,
        netif->crcs + netif->oerrs + netif->frames + netif->overflows +
        netif->buffs + netif->soverflows);
}
```

⟨*global* etherdevtab *(arm)* 326b⟩≡ (326c)

```
Dev etherdevtab = {
    .dc = 'l',
    .name = "ether",

    .reset = etherreset,
    .init = devinit,
    .shutdown = ethershutdown,
    .attach = etherattach,
    .walk = etherwalk,
    .stat = etherstat,
    .open = etheropen,
    .create = ethercreate,
    .close = etherclose,
    .read = etherread,
    .bread = etherbread,
    .write = etherwrite,
    .bwrite = etherbwrite,
    .remove = devremove,
    .wstat = etherwstat,
};
```

⟨kernel/network/arm/devether.c 326c⟩≡

```
#include "u.h"
#include "../port/lib.h"
#include "../port/error.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

#include "../port/netif.h"
#include "../port/etherif.h"

extern int archether(unsigned ctlno, Ether *ether);
```

⟨*global* etherxx *(arm)* 317a⟩

⟨*function* etherattach *(arm)* 317b⟩

⟨*function* etherwalk *(arm)* 317c⟩

⟨*function* etherstat *(arm)* 318a⟩

⟨*function* etheropen *(arm)* 318b⟩

⟨*function* ethercreate *(arm)* 318c⟩

⟨*function* etherclose *(arm)* 318d⟩

⟨*function* etherread *(arm)* 318e⟩

⟨*function* etherbread *(arm)* 319a⟩

⟨*function* etherwstat *(arm)* 319b⟩

⟨*function* etherrtrace *(arm)* 319c⟩

⟨*function* etheriq *(arm)* 319d⟩

⟨*function* etheroq *(arm)* 321a⟩

⟨*function* etherwrite *(arm)* 321b⟩

⟨*function* etherbwrite *(arm)* 322⟩

⟨*global* cards *(arm)* 323a⟩

⟨*function* addethercard *(arm)* 323b⟩

⟨*function* parseether *(arm)* 323c⟩

⟨*function* etherreset *(arm)* 323d⟩

⟨*function* ethershutdown *(arm)* 325a⟩

⟨*constant* POLY *(arm)* 325b⟩

⟨*function* ethercrc *(arm)* 325c⟩

⟨*function* dumpoq *(arm)* 325d⟩

⟨*function* dumpnetif *(arm)* 326a⟩

⟨*global* etherdevtab *(arm)* 326b⟩

I.8 kernel/network/ip/

I.8.1 kernel/network/ip/devip.c

⟨*function* ipreset 327⟩≡ (331c)

```
static void
ipreset(void)
```

```
{
    nullmediumlink();
    pktmediumlink();

    fmtinstall('i', eipfmt);
    fmtinstall('I', eipfmt);
    fmtinstall('E', eipfmt);
    fmtinstall('V', eipfmt);
    fmtinstall('M', eipfmt);
}
```

Uses `eipfmt()` 186, `nullmediumlink()` 216e, and `pktmediumlink()` 369a.

⟨*function* `ipstat` 328a⟩≡ (331c)

```
static int
ipstat(Chan* c, uchar* db, int n)
{
    return devstat(c, db, n, nil, 0, ipgen);
}
```

Uses `ipgen()` 45b.

⟨*function* `incoming` 328b⟩≡ (331c)

```
static int
incoming(void* arg)
{
    Conv *conv;

    conv = arg;
    return conv->incall != nil;
}
```

⟨*function* `ipwstat` 328c⟩≡ (331c)

```
static int
ipwstat(Chan *c, uchar *dp, int n)
{
    Dir d;
    Conv *cv;
    Fs *f;
    Proto *p;

    f = ipfs[c->dev];
    switch(TYPE(c->qid)) {
    default:
        error(Eperm);
        break;
    case Qctl:
    case Qdata:
        break;
    }

    n = convM2D(dp, n, &d, nil);
    if(n > 0){
        p = f->p[PROTO(c->qid)];
        cv = p->conv[CONV(c->qid)];
        if(!iseve() && strcmp(ATTACHER(c), cv->owner) != 0)
            error(Eperm);
        if(d.uid[0])
            kstrdup(&cv->owner, d.uid);
        cv->perm = d.mode & 0777;
    }
    return n;
```

```
  }
```
Uses ATTACHER-250 41b, CONV-246 35d, PROTO-247 35c, Qctl-228 34i, Qdata-229 34i, TYPE-245 35e, and ipfs 29f.

⟨*enum* _anon_ (kernel/network/ip/devip.c)2 329a⟩≡ (331c)
```
  enum
  {
      ⟨constant Statelen 64c⟩
  };
```

⟨*function* ipbread 329b⟩≡ (331c)
```
  static Block*
  ipbread(Chan* ch, long n, ulong offset)
  {
      Conv *c;
      Proto *x;
      Fs *f;

      switch(TYPE(ch->qid)){
      case Qdata:
          f = ipfs[ch->dev];
          x = f->p[PROTO(ch->qid)];
          c = x->conv[CONV(ch->qid)];
          return qbread(c->rq, n);
      default:
          return devbread(ch, n, offset);
      }
  }
```
Uses CONV-246 35d, PROTO-247 35c, Qdata-229 34i, TYPE-245 35e, and ipfs 29f.

⟨*function* setladdr 329c⟩≡ (331c)
```
  /*
   *  set local address to be that of the ifc closest to remote address
   */
  static void
  setladdr(Conv* c)
  {
      findlocalip(c->p->f, c->laddr, c->raddr);
  }
```
Uses findlocalip() 379.

⟨*function* setluniqueport 329d⟩≡ (331c)
```
  /*
   *  set a local port making sure the quad of raddr,rport,laddr,lport is unique
   */
  char*
  setluniqueport(Conv* c, int lport)
  {
      Proto *p;
      Conv *xp;
      int x;

      p = c->p;

      qlock(p);
      for(x = 0; x < p->nc; x++){
          xp = p->conv[x];
          if(xp == nil)
              break;
          if(xp == c)
```

```
            continue;
        if((xp->state == Connected || xp->state == Announced)
        && xp->lport == lport
        && xp->rport == c->rport
        && ipcmp(xp->raddr, c->raddr) == 0
        && ipcmp(xp->laddr, c->laddr) == 0){
            qunlock(p);
            return "address in use";
        }
    }
    c->lport = lport;
    qunlock(p);
    return nil;
}
```

Uses `Announced` 34b, `Connected` 34b, and `ipcmp` 23d.

⟨*function* `lportinuse` 330a⟩≡ (331c)

```
/*
 * is lport in use by anyone?
 */
static int
lportinuse(Proto *p, ushort lport)
{
    int x;

    for(x = 0; x < p->nc && p->conv[x]; x++)
        if(p->conv[x]->lport == lport)
            return 1;
    return 0;
}
```

⟨*function* `ipbwrite` 330b⟩≡ (331c)

```
static long
ipbwrite(Chan* ch, Block* bp, ulong offset)
{
    Conv *c;
    Proto *x;
    Fs *f;
    int n;

    switch(TYPE(ch->qid)){
    case Qdata:
        f = ipfs[ch->dev];
        x = f->p[PROTO(ch->qid)];
        c = x->conv[CONV(ch->qid)];

        if(c->wq == nil)
            error(Eperm);

        if(bp->next)
            bp = concatblock(bp);
        n = BLEN(bp);
        qbwrite(c->wq, bp);
        return n;
    default:
        return devbwrite(ch, bp, offset);
    }
}
```

Uses `CONV-246` 35d, `PROTO-247` 35c, `Qdata-229` 34i, `TYPE-245` 35e, and `ipfs` 29f.

⟨*function* Fsrcvpcolx 331a⟩≡ (331c)
```
Proto*
Fsrcvpcolx(Fs *f, uchar proto)
{
    return f->t2p[proto];
}
```

⟨*function* ndbwrite 331b⟩≡ (331c)
```
long
ndbwrite(Fs *f, char *a, ulong off, int n)
{
    if(off > strlen(f->ndb))
        error(Eio);
    if(off+n >= sizeof(f->ndb))
        error(Eio);
    memmove(f->ndb+off, a, n);
    f->ndb[off+n] = 0;
    f->ndbvers++;
    f->ndbmtime = seconds();
    return n;
}
```

⟨kernel/network/ip/devip.c 331c⟩≡
```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"
#include    "../ip/ip.h"

⟨enum qid((kernel/network/ip/devip.c)) 34i⟩

⟨enum misc((kernel/network/ip/devip.c)) 35a⟩

⟨macro TYPE 35e⟩
⟨macro CONV 35d⟩
⟨macro PROTO 35c⟩
⟨macro QID 35b⟩

⟨global network 46a⟩

⟨global fslock 30a⟩
⟨global ipfs 29f⟩
//Queue *qlog;

extern  void nullmediumlink(void);
extern  void pktmediumlink(void);
long ndbwrite(Fs *f, char *a, ulong off, int n);
Conv*   Fsprotoclone(Proto*, char*);
char*   Fsstdbind(Conv*, char**, int);
void    closeconv(Conv*);

⟨function ip3gen 48a⟩

⟨function ip2gen 47b⟩

⟨function ip1gen 46c⟩

⟨function ipgen 45b⟩
```

⟨*function* ipreset 327⟩

⟨*function* ipgetfs 40b⟩

⟨*function* newipaux 41c⟩

⟨*macro* ATTACHER 41b⟩

⟨*function* ipattach 40a⟩

⟨*function* ipwalk 45a⟩

⟨*function* ipstat 328a⟩

⟨*function* incoming 328b⟩

⟨*global* m2p 48c⟩

⟨*function* ipopen 49a⟩

⟨*function* ipcreate 44b⟩

⟨*function* ipremove 44c⟩

⟨*function* ipwstat 328c⟩

⟨*function* closeconv 50c⟩

⟨*function* ipclose 50a⟩

⟨*enum* _anon_ (kernel/network/ip/devip.c)2 329a⟩

⟨*function* ipread 51a⟩

⟨*function* ipbread 329b⟩

⟨*function* setladdr 329c⟩

⟨*function* setluniqueport 329d⟩

⟨*function* lportinuse 330a⟩

⟨*function* setlport 58f⟩

⟨*function* setladdrport 59⟩

⟨*function* setraddrport 60⟩

⟨*function* Fsstdconnect 56b⟩
⟨*function* connected 56a⟩
⟨*function* connectctlmsg 55c⟩

⟨*function* Fsstdannounce 58b⟩

⟨*function* announced 58a⟩
⟨*function* announcectlmsg 57c⟩

⟨*function* Fsstdbind 61c⟩

⟨*function* bindctlmsg 61b⟩

⟨*function* tosctlmsg 81c⟩

⟨*function* ttlctlmsg 81f⟩

⟨*function* ipwrite 51c⟩

⟨*function* ipbwrite 330b⟩

⟨*global* ipdevtab 44a⟩

⟨*function* Fsproto 68a⟩

⟨*function* Fsprotoclone 52b⟩

⟨*function* Fsconnected 57a⟩

⟨*function* Fsrcvpcol 99d⟩

⟨*function* Fsrcvpcolx 331a⟩

⟨*function* Fsnewcall 144⟩

⟨*function* ndbwrite 331b⟩

⟨*function* scalednconv 151d⟩

I.8.2 kernel/network/ip/arp.c

⟨*enum* _anon_ (kernel/network/ip/arp.c) 333a⟩≡ (341b)

```
/*
 *  address resolution tables
 */

enum
{
    NHASH       = (1<<6),
    NCACHE      = 256,

    AOK     = 1,
    AWAIT       = 2,
};
```

⟨*global* arpstate 333b⟩≡ (341b)

```
char *arpstate[] =
{
    "UNUSED",
    "OK",
    "WAIT",
};
```

⟨*global* Ebadarp 333c⟩≡ (341b)

```
char *Ebadarp = "bad arp";
```

Uses Ebadarp 333c.

⟨*macro* haship 333d⟩≡ (341b)

```
#define haship(s) ((s)[IPaddrlen-1]%NHASH)
```

⟨*global* ReTransTimer 334a⟩≡ (341b)
```
int     ReTransTimer = RETRANS_TIMER;
```
Uses RETRANS_TIMER 498e and ReTransTimer 334a.

⟨*function* newarp6 334b⟩≡ (341b)
```
/*
 *  create a new arp entry for an ip address.
 */
static Arpent*
newarp6(Arp *arp, uchar *ip, Ipifc *ifc, int addrxt)
{
    uint t;
    Block *next, *xp;
    Arpent *a, *e, *f, **l;
    Medium *m = ifc->m;
    int empty;

    /* find oldest entry */
    e = &arp->cache[NCACHE];
    a = arp->cache;
    t = a->utime;
    for(f = a; f < e; f++){
        if(f->utime < t){
            t = f->utime;
            a = f;
        }
    }

    /* dump waiting packets */
    xp = a->hold;
    a->hold = nil;

    if(isv4(a->ip)){
        while(xp){
            next = xp->list;
            freeblist(xp);
            xp = next;
        }
    }
    else { /* queue icmp unreachable for rxmitproc later on, w/o arp lock */
        if(xp){
            if(arp->dropl == nil)
                arp->dropf = xp;
            else
                arp->dropl->list = xp;

            for(next = xp->list; next; next = next->list)
                xp = next;
            arp->dropl = xp;
            wakeup(&arp->rxmtq);
        }
    }

    /* take out of current chain */
    l = &arp->hash[haship(a->ip)];
    for(f = *l; f; f = f->hash){
        if(f == a){
            *l = a->hash;
            break;
        }
```

```
        l = &f->hash;
    }

    /* insert into new chain */
    l = &arp->hash[haship(ip)];
    a->hash = *l;
    *l = a;

    memmove(a->ip, ip, sizeof(a->ip));
    a->utime = NOW;
    a->ctime = 0;
    a->type = m;

    a->rtime = NOW + ReTransTimer;
    a->rxtsrem = MAX_MULTICAST_SOLICIT;
    a->ifc = ifc;
    a->ifcid = ifc->ifcid;

    /* put to the end of re-transmit chain; addrxt is 0 when isv4(a->ip) */
    if(!ipismulticast(a->ip) && addrxt){
        l = &arp->rxmt;
        empty = (*l==nil);

        for(f = *l; f; f = f->nextrxt){
            if(f == a){
                *l = a->nextrxt;
                break;
            }
            l = &f->nextrxt;
        }
        for(f = *l; f; f = f->nextrxt){
            l = &f->nextrxt;
        }
        *l = a;
        if(empty)
            wakeup(&arp->rxmtq);
    }

    a->nextrxt = nil;

    return a;
}
```

Uses `MAX_MULTICAST_SOLICIT` 498e, `NCACHE-116` 333a, `NOW` 234a, `ReTransTimer` 334a, `haship-119` 333d, `ipismulticast()` 381c, and `isv4()` 21b.

⟨*function* `cleanarpent` 335⟩≡ (341b)

```
/* called with arp qlocked */

void
cleanarpent(Arp *arp, Arpent *a)
{
    Arpent *f, **l;

    a->utime = 0;
    a->ctime = 0;
    a->type = 0;
    a->state = 0;

    /* take out of current chain */
    l = &arp->hash[haship(a->ip)];
```

```
    for(f = *l; f; f = f->hash){
        if(f == a){
            *l = a->hash;
            break;
        }
        l = &f->hash;
    }

    /* take out of re-transmit chain */
    l = &arp->rxmt;
    for(f = *l; f; f = f->nextrxt){
        if(f == a){
            *l = a->nextrxt;
            break;
        }
        l = &f->nextrxt;
    }
    a->nextrxt = nil;
    a->hash = nil;
    a->hold = nil;
    a->last = nil;
    a->ifc = nil;
}
```

Uses haship-119 333d.

⟨*function* arpget 336⟩≡ (341b)

```
/*
 *  fill in the media address if we have it.  Otherwise return an
 *  Arpent that represents the state of the address resolution FSM
 *  for ip.  Add the packet to be sent onto the list of packets
 *  waiting for ip->mac to be resolved.
 */
Arpent*
arpget(Arp *arp, Block *bp, int version, Ipifc *ifc, uchar *ip, uchar *mac)
{
    int hash;
    Arpent *a;
    Medium *type = ifc->m;
    ipaddr v6ip;

    if(version == V4){
        v4tov6(v6ip, ip);
        ip = v6ip;
    }

    qlock(arp);
    hash = haship(ip);
    for(a = arp->hash[hash]; a; a = a->hash){
        if(memcmp(ip, a->ip, sizeof(a->ip)) == 0)
        if(type == a->type)
            break;
    }

    if(a == nil){
        a = newarp6(arp, ip, ifc, (version != V4));
        a->state = AWAIT;
    }
    a->utime = NOW;
    if(a->state == AWAIT){
        if(bp != nil){
```

```
            if(a->hold)
                a->last->list = bp;
            else
                a->hold = bp;
            a->last = bp;
            bp->list = nil;
        }
        return a;       /* return with arp qlocked */
    }

    memmove(mac, a->mac, a->type->maclen);

    /* remove old entries */
    if(NOW - a->ctime > 15*60*1000)
        cleanarpent(arp, a);

    qunlock(arp);
    return nil;
}
```

Uses AWAIT-118 333a, NOW 234a, V4 21g, cleanarpent() 335, haship-119 333d, newarp6() 334b, and v4tov6() 21c.

⟨*function* arprelease 337a⟩≡ (341b)

```
/*
 * called with arp locked
 */
void
arprelease(Arp *arp, Arpent*)
{
    qunlock(arp);
}
```

⟨*function* arpresolve 337b⟩≡ (341b)

```
/*
 * Copy out the mac address from the Arpent.  Return the
 * block waiting to get sent to this mac address.
 *
 * called with arp locked
 */
Block*
arpresolve(Arp *arp, Arpent *a, Medium *type, uchar *mac)
{
    Block *bp;
    Arpent *f, **l;

    if(!isv4(a->ip)){
        l = &arp->rxmt;
        for(f = *l; f; f = f->nextrxt){
            if(f == a){
                *l = a->nextrxt;
                break;
            }
            l = &f->nextrxt;
        }
    }

    memmove(a->mac, mac, type->maclen);
    a->type = type;
    a->state = AOK;
    a->utime = NOW;
    bp = a->hold;
```

```
    a->hold = nil;
    qunlock(arp);

    return bp;
}
```

Uses AOK-117 333a, NOW 234a, and isv4() 21b.

⟨*function* arpenter 338⟩≡ (341b)

```
void
arpenter(Fs *fs, int version, uchar *ip, uchar *mac, int n, int refresh)
{
    Arp *arp;
    Route *r;
    Arpent *a, *f, **l;
    Ipifc *ifc;
    Medium *type;
    Block *bp, *next;
    ipaddr v6ip;

    arp = fs->arp;

    if(n != 6){
//      print("arp: len = %d\n", n);
        return;
    }

    switch(version){
    case V4:
        r = v4lookup(fs, ip, nil);
        v4tov6(v6ip, ip);
        ip = v6ip;
        break;
    case V6:
        r = v6lookup(fs, ip, nil);
        break;
    default:
        panic("arpenter: version %d", version);
        return; /* to supress warnings */
    }

    if(r == nil){
//      print("arp: no route for entry\n");
        return;
    }

    ifc = r->ifc;
    type = ifc->m;

    qlock(arp);
    for(a = arp->hash[haship(ip)]; a; a = a->hash){
        if(a->type != type || (a->state != AWAIT && a->state != AOK))
            continue;

        if(ipcmp(a->ip, ip) == 0){
            a->state = AOK;
            memmove(a->mac, mac, type->maclen);

            if(version == V6){
                /* take out of re-transmit chain */
                l = &arp->rxmt;
```

```
                for(f = *l; f; f = f->nextrxt){
                    if(f == a){
                        *l = a->nextrxt;
                        break;
                    }
                    l = &f->nextrxt;
                }
            }

            a->ifc = ifc;
            a->ifcid = ifc->ifcid;
            bp = a->hold;
            a->hold = nil;
            if(version == V4)
                ip += IPv4off;
            a->utime = NOW;
            a->ctime = a->utime;
            qunlock(arp);

            while(bp){
                next = bp->list;
                if(ifc != nil){
                    if(waserror()){
                        runlock(ifc);
                        nexterror();
                    }
                    rlock(ifc);
                    if(ifc->m != nil)
                        ifc->m->bwrite(ifc, bp, version, ip);
                    else
                        freeb(bp);
                    runlock(ifc);
                    poperror();
                } else
                    freeb(bp);
                bp = next;
            }
            return;
        }
    }

    if(refresh == 0){
        a = newarp6(arp, ip, ifc, 0);
        a->state = AOK;
        a->type = type;
        a->ctime = NOW;
        memmove(a->mac, mac, type->maclen);
    }

    qunlock(arp);
}
```

Uses `AOK-117` 333a, `AWAIT-118` 333a, `IPv4off` 21a, `NOW` 234a, `V4` 21g, `V6` 21g, `haship-119` 333d, `ipcmp` 23d, `newarp6()` 334b, `v4lookup()` 112a, `v4tov6()` 21c, and `v6lookup()` 508b.

⟨*enum* `_anon_ (kernel/network/ip/arp.c)2` 339⟩≡ (341b)

```
enum
{
    Alinelen=   90,
};
```

⟨*global* aformat 340a⟩≡ (341b)

```
char *aformat = "%-6.6s %-8.8s %-40.40I %-32.32s\n";
```

Uses aformat 340a.

⟨*function* convmac 340b⟩≡ (341b)

```
static void
convmac(char *p, char *ep, uchar *mac, int n)
{
    while(n-- > 0)
        p = seprint(p, ep, "%2.2ux", *mac++);
}
```

⟨*function* rxmitsols 340c⟩≡ (341b)

```
extern int
rxmitsols(Arp *arp)
{
    uint sflag;
    Block *next, *xp;
    Arpent *a, *b, **l;
    Fs *f;
    ipaddr ipsrc;
    Ipifc *ifc = nil;
    long nrxt;

    qlock(arp);
    f = arp->f;

    a = arp->rxmt;
    if(a==nil){
        nrxt = 0;
        goto dodrops;       /* return nrxt; */
    }
    nrxt = a->rtime - NOW;
    if(nrxt > 3*ReTransTimer/4)
        goto dodrops;       /* return nrxt; */

    for(; a; a = a->nextrxt){
        ifc = a->ifc;
        assert(ifc != nil);
        if((a->rxtsrem <= 0) || !(canrlock(ifc)) || (a->ifcid != ifc->ifcid)){
            xp = a->hold;
            a->hold = nil;

            if(xp){
                if(arp->dropl == nil)
                    arp->dropf = xp;
                else
                    arp->dropl->list = xp;
            }

            cleanarpent(arp, a);
        }
        else
            break;
    }
    if(a == nil)
        goto dodrops;


    qunlock(arp);   /* for icmpns */
```

```
    if((sflag = ipv6anylocal(ifc, ipsrc)) != SRC_UNSPEC)
        icmpns(f, ipsrc, sflag, a->ip, TARG_MULTI, ifc->mac);

    runlock(ifc);
    qlock(arp);

    /* put to the end of re-transmit chain */
    l = &arp->rxmt;
    for(b = *l; b; b = b->nextrxt){
        if(b == a){
            *l = a->nextrxt;
            break;
        }
        l = &b->nextrxt;
    }
    for(b = *l; b; b = b->nextrxt){
        l = &b->nextrxt;
    }
    *l = a;
    a->rxtsrem--;
    a->nextrxt = nil;
    a->rtime = NOW + ReTransTimer;

    a = arp->rxmt;
    if(a==nil)
        nrxt = 0;
    else
        nrxt = a->rtime - NOW;

dodrops:
    xp = arp->dropf;
    arp->dropf = nil;
    arp->dropl = nil;
    qunlock(arp);

    for(; xp; xp = next){
        next = xp->list;
        icmphostunr(f, ifc, xp, Icmp6_adr_unreach, 1);
    }

    return nrxt;

}
```

Uses `Icmp6_adr_unreach` 498e, `NOW` 234a, `ReTransTimer` 334a, `SRC_UNSPEC` 498e, `TARG_MULTI` 498e, `cleanarpent()` 335, `icmphostunr()` 476, `icmpns()` 474b, and `ipv6anylocal()` 512b.

⟨*function* `rxready` 341a⟩≡ (341b)

```
static int
rxready(void *v)
{
    Arp *arp = (Arp *) v;
    int x;

    x = ((arp->rxmt != nil) || (arp->dropf != nil));

    return x;
}
```

⟨`kernel/network/ip/arp.c` 341b⟩≡

```
#include "u.h"
```

```
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

#include "ip.h"
#include "ipv6.h"

⟨enum _anon_ (kernel/network/ip/arp.c) 333a⟩

⟨global arpstate 333b⟩

⟨struct Arp 106b⟩

⟨global Ebadarp 333c⟩

⟨macro haship 333d⟩

⟨global ReTransTimer 334a⟩

static void     rxmitproc(void *v);

⟨function arpinit 107b⟩

⟨function newarp6 334b⟩

⟨function cleanarpent 335⟩

⟨function arpget 336⟩

⟨function arprelease 337a⟩

⟨function arpresolve 337b⟩

⟨function arpenter 338⟩

⟨function arpwrite 109b⟩

⟨enum _anon_ (kernel/network/ip/arp.c)2 339⟩

⟨global aformat 340a⟩

⟨function convmac 340b⟩

⟨function arpread 108d⟩

⟨function rxmitsols 340c⟩

⟨function rxready 341a⟩

⟨function rxmitproc 107c⟩
```

I.8.3 kernel/network/ip/icmp.c

⟨*struct* Icmp 342⟩≡ (352)

```
typedef struct Icmp {
    // ip header
```

```
    uchar   vihl;       /* Version and header length */
    uchar   tos;        /* Type of service */
    uchar   length[2];  /* packet length */
    uchar   id[2];      /* Identification */
    uchar   frag[2];    /* Fragment information */
    uchar   ttl;        /* Time to live */
    uchar   proto;      /* Protocol */
    uchar   ipcksum[2]; /* Header checksum */
    uchar   src[4];     /* Ip source */
    uchar   dst[4];     /* Ip destination */

    // ICMP specifics
    uchar   type;
    uchar   code;
    uchar   cksum[2];
    uchar   icmpid[2];
    uchar   seq[2];
    uchar   data[1];
} Icmp;
```

Uses `Icmp` 342.

⟨*enum* `_anon_ (kernel/network/ip/icmp.c)` 343a⟩≡ (352)

```
enum {          /* Packet Types */
    EchoReply   = 0,
    Unreachable = 3,
    SrcQuench   = 4,
    Redirect    = 5,
    EchoRequest = 8,
    TimeExceed  = 11,
    InParmProblem   = 12,
    Timestamp   = 13,
    TimestampReply  = 14,
    InfoRequest = 15,
    InfoReply   = 16,
    AddrMaskRequest = 17,
    AddrMaskReply   = 18,

    Maxtype     = 18,
};
```

⟨*enum* `_anon_ (kernel/network/ip/icmp.c)2` 343b⟩≡ (352)

```
enum
{
    MinAdvise   = 24,   /* minimum needed for us to advise another protocol */
};
```

⟨*global* `icmpnames` 343c⟩≡ (352)

```
char *icmpnames[Maxtype+1] =
{
[EchoReply]     "EchoReply",
[Unreachable]       "Unreachable",
[SrcQuench]     "SrcQuench",
[Redirect]      "Redirect",
[EchoRequest]       "EchoRequest",
[TimeExceed]        "TimeExceed",
[InParmProblem]     "InParmProblem",
[Timestamp]     "Timestamp",
[TimestampReply]    "TimestampReply",
[InfoRequest]       "InfoRequest",
[InfoReply]     "InfoReply",
```

```
[AddrMaskRequest]   "AddrMaskRequest",
[AddrMaskReply  ]   "AddrMaskReply  ",
};
```
Uses Maxtype-201 343a.

⟨*enum* _anon_ (kernel/network/ip/icmp.c)3 344a⟩≡ (352)
```
enum {
    IP_ICMPPROTO    = 1,
    ICMP_IPSIZE = 20,
    ICMP_HDRSIZE    = 8,
};
```

⟨*enum* _anon_ (kernel/network/ip/icmp.c)4 344b⟩≡ (352)
```
enum
{
    InMsgs,
    InErrors,
    OutMsgs,
    CsumErrs,
    LenErrs,
    HlenErrs,

    Nstats,
};
```

⟨*global* statnames*(*(kernel/network/ip/icmp.c)*)* 344c⟩≡ (352)
```
static char *statnames[Nstats] =
{
[InMsgs]    "InMsgs",
[InErrors]  "InErrors",
[OutMsgs]   "OutMsgs",
[CsumErrs]  "CsumErrs",
[LenErrs]   "LenErrs",
[HlenErrs]  "HlenErrs",
};
```
Uses Nstats-212 344b.

⟨*struct* Icmppriv 344d⟩≡ (352)
```
struct Icmppriv
{
    ulong   stats[Nstats];

    /* message counts */
    ulong   in[Maxtype+1];
    ulong   out[Maxtype+1];
};
```
Uses Maxtype-201 343a and Nstats-212 344b.

⟨*function* icmpcreate 344e⟩≡ (352)
```
static void
icmpcreate(Conv *c)
{
    c->rq = qopen(64*1024, Qmsg, 0, c);
    c->wq = qbypass(icmpkick, c);
}
```
Uses icmpkick() 345e.

⟨*function* icmpconnect 345a⟩≡ (352)

```
extern char*
icmpconnect(Conv *c, char **argv, int argc)
{
    char *e;

    e = Fsstdconnect(c, argv, argc);
    if(e != nil)
        return e;
    Fsconnected(c, e);

    return nil;
}
```

Uses `Fsstdconnect()` 56b.

⟨*function* icmpstate 345b⟩≡ (352)

```
extern int
icmpstate(Conv *c, char *state, int n)
{
    USED(c);
    return snprint(state, n, "%s qin %d qout %d\n",
        "Datagram",
        c->rq ? qlen(c->rq) : 0,
        c->wq ? qlen(c->wq) : 0
    );
}
```

⟨*function* icmpannounce 345c⟩≡ (352)

```
extern char*
icmpannounce(Conv *c, char **argv, int argc)
{
    char *e;

    e = Fsstdannounce(c, argv, argc);
    if(e != nil)
        return e;
    Fsconnected(c, nil);

    return nil;
}
```

Uses `Fsstdannounce()` 58b.

⟨*function* icmpclose 345d⟩≡ (352)

```
extern void
icmpclose(Conv *c)
{
    qclose(c->rq);
    qclose(c->wq);
    ipmove(c->laddr, IPnoaddr);
    ipmove(c->raddr, IPnoaddr);
    c->lport = 0;
}
```

Uses `IPnoaddr` 23b and `ipmove` 23c.

⟨*function* icmpkick 345e⟩≡ (352)

```
static void
icmpkick(void *x, Block *bp)
{
    Conv *c = x;
```

```
    Icmp *p;
    Icmppriv *ipriv;

    if(bp == nil)
        return;

    if(blocklen(bp) < ICMP_IPSIZE + ICMP_HDRSIZE){
        freeblist(bp);
        return;
    }
    p = (Icmp *)(bp->rp);
    p->vihl = IP_VER4;
    ipriv = c->p->priv;
    if(p->type <= Maxtype)
        ipriv->out[p->type]++;

    v6tov4(p->dst, c->raddr);
    v6tov4(p->src, c->laddr);
    p->proto = IP_ICMPPROTO;
    hnputs(p->icmpid, c->lport);
    memset(p->cksum, 0, sizeof(p->cksum));
    hnputs(p->cksum, ptclcsum(bp, ICMP_IPSIZE, blocklen(bp) - ICMP_IPSIZE));
    ipriv->stats[OutMsgs]++;
    ipoput4(c->p->f, bp, 0, c->ttl, c->tos, nil);
}
```

Uses ICMP_HDRSIZE-205 344a, ICMP_IPSIZE-204 344a, IP_ICMPPROTO-203 344a, IP_VER4 94a, Maxtype-201 343a, OutMsgs-208 344b, ipoput4() 93, ptclcsum() 90d, and v6tov4() 21d.

⟨*function* icmpttlexceeded 346a⟩≡ (352)

```
extern void
icmpttlexceeded(Fs *f, uchar *ia, Block *bp)
{
    Block   *nbp;
    Icmp    *p, *np;

    p = (Icmp *)bp->rp;

    netlog(f, Logicmp, "sending icmpttlexceeded -> %V\n", p->src);
    nbp = allocb(ICMP_IPSIZE + ICMP_HDRSIZE + ICMP_IPSIZE + 8);
    nbp->wp += ICMP_IPSIZE + ICMP_HDRSIZE + ICMP_IPSIZE + 8;
    np = (Icmp *)nbp->rp;
    np->vihl = IP_VER4;
    memmove(np->dst, p->src, sizeof(np->dst));
    v6tov4(np->src, ia);
    memmove(np->data, bp->rp, ICMP_IPSIZE + 8);
    np->type = TimeExceed;
    np->code = 0;
    np->proto = IP_ICMPPROTO;
    hnputs(np->icmpid, 0);
    hnputs(np->seq, 0);
    memset(np->cksum, 0, sizeof(np->cksum));
    hnputs(np->cksum, ptclcsum(nbp, ICMP_IPSIZE, blocklen(nbp) - ICMP_IPSIZE));
    ipoput4(f, nbp, 0, MAXTTL, DFLTTOS, nil);

}
```

Uses DFLTTOS 232b, ICMP_HDRSIZE-205 344a, ICMP_IPSIZE-204 344a, IP_ICMPPROTO-203 344a, IP_VER4 94a, Logicmp 233d, MAXTTL 81g, TimeExceed-193 343a, ipoput4() 93, netlog() 389a, ptclcsum() 90d, and v6tov4() 21d.

⟨*function* icmpunreachable 346b⟩≡ (352)

```
static void
```

```
icmpunreachable(Fs *f, Block *bp, int code, int seq)
{
    Block   *nbp;
    Icmp    *p, *np;
    int i;
    ipaddr   addr;

    p = (Icmp *)bp->rp;

    /* only do this for unicast sources and destinations */
    v4tov6(addr, p->dst);
    i = ipforme(f, addr);
    if((i&Runi) == 0)
        return;
    v4tov6(addr, p->src);
    i = ipforme(f, addr);
    if(i != 0 && (i&Runi) == 0)
        return;

    netlog(f, Logicmp, "sending icmpnoconv -> %V\n", p->src);
    nbp = allocb(ICMP_IPSIZE + ICMP_HDRSIZE + ICMP_IPSIZE + 8);
    nbp->wp += ICMP_IPSIZE + ICMP_HDRSIZE + ICMP_IPSIZE + 8;
    np = (Icmp *)nbp->rp;
    np->vihl = IP_VER4;
    memmove(np->dst, p->src, sizeof(np->dst));
    memmove(np->src, p->dst, sizeof(np->src));
    memmove(np->data, bp->rp, ICMP_IPSIZE + 8);
    np->type = Unreachable;
    np->code = code;
    np->proto = IP_ICMPPROTO;
    hnputs(np->icmpid, 0);
    hnputs(np->seq, seq);
    memset(np->cksum, 0, sizeof(np->cksum));
    hnputs(np->cksum, ptclcsum(nbp, ICMP_IPSIZE, blocklen(nbp) - ICMP_IPSIZE));
    ipoput4(f, nbp, 0, MAXTTL, DFLTTOS, nil);
}
```

Uses `DFLTTOS` 232b, `ICMP_HDRSIZE-205` 344a, `ICMP_IPSIZE-204` 344a, `IP_ICMPPROTO-203` 344a, `IP_VER4` 94a, `Logicmp` 233d, `MAXTTL` 81g, `Runi` 38e, `Unreachable-189` 343a, `ipforme()` 125e, `ipoput4()` 93, `netlog()` 389a, `ptclcsum()` 90d, and `v4tov6()` 21c.

⟨*function* `icmpnoconv` 347a⟩≡ (352)

```
extern void
icmpnoconv(Fs *f, Block *bp)
{
    icmpunreachable(f, bp, 3, 0);
}
```

Uses `icmpunreachable()` 346b.

⟨*function* `icmpcantfrag` 347b⟩≡ (352)

```
extern void
icmpcantfrag(Fs *f, Block *bp, int mtu)
{
    icmpunreachable(f, bp, 4, mtu);
}
```

Uses `icmpunreachable()` 346b.

⟨*function* `goticmpkt` 347c⟩≡ (352)

```
static void
goticmpkt(Proto *icmp, Block *bp)
```

```
{
    Conv    **c, *s;
    Icmp    *p;
    ipaddr  dst;
    ushort  recid;

    p = (Icmp *) bp->rp;
    v4tov6(dst, p->src);
    recid = nhgets(p->icmpid);

    for(c = icmp->conv; *c; c++) {
        s = *c;
        if(s->lport == recid)
        if(ipcmp(s->raddr, dst) == 0){
            bp = concatblock(bp);
            if(bp != nil)
                qpass(s->rq, bp);
            return;
        }
    }
    freeblist(bp);
}
```
Uses ipcmp 23d and v4tov6() 21c.

⟨*function* mkechoreply 348a⟩≡ (352)
```
static Block *
mkechoreply(Block *bp)
{
    Icmp    *q;
    uchar   ip[4];

    q = (Icmp *)bp->rp;
    q->vihl = IP_VER4;
    memmove(ip, q->src, sizeof(q->dst));
    memmove(q->src, q->dst, sizeof(q->src));
    memmove(q->dst, ip,  sizeof(q->dst));
    q->type = EchoReply;
    memset(q->cksum, 0, sizeof(q->cksum));
    hnputs(q->cksum, ptclcsum(bp, ICMP_IPSIZE, blocklen(bp) - ICMP_IPSIZE));

    return bp;
}
```
Uses EchoReply-188 343a, ICMP_IPSIZE-204 344a, IP_VER4 94a, and ptclcsum() 90d.

⟨*global* unreachcode 348b⟩≡ (352)
```
static char *unreachcode[] =
{
[0] "net unreachable",
[1] "host unreachable",
[2] "protocol unreachable",
[3] "port unreachable",
[4] "fragmentation needed and DF set",
[5] "source route failed",
};
```

⟨*function* icmpiput 348c⟩≡ (352)
```
static void
icmpiput(Proto *icmp, Ipifc*, Block *bp)
{
    int n, iplen;
```

```
Icmp    *p;
Block   *r;
Proto   *pr;
char    *msg;
char    m2[128];
Icmppriv *ipriv;

ipriv = icmp->priv;

ipriv->stats[InMsgs]++;

p = (Icmp *)bp->rp;
netlog(icmp->f, Logicmp, "icmpiput %s (%d) %d\n",
    (p->type < nelem(icmpnames)? icmpnames[p->type]: ""),
    p->type, p->code);
n = blocklen(bp);
if(n < ICMP_IPSIZE+ICMP_HDRSIZE){
    ipriv->stats[InErrors]++;
    ipriv->stats[HlenErrs]++;
    netlog(icmp->f, Logicmp, "icmp hlen %d\n", n);
    goto raise;
}
iplen = nhgets(p->length);
if(iplen > n){
    ipriv->stats[LenErrs]++;
    ipriv->stats[InErrors]++;
    netlog(icmp->f, Logicmp, "icmp length %d\n", iplen);
    goto raise;
}
if(ptclcsum(bp, ICMP_IPSIZE, iplen - ICMP_IPSIZE)){
    ipriv->stats[InErrors]++;
    ipriv->stats[CsumErrs]++;
    netlog(icmp->f, Logicmp, "icmp checksum error\n");
    goto raise;
}
if(p->type <= Maxtype)
    ipriv->in[p->type]++;

switch(p->type) {
case EchoRequest:
    if (iplen < n)
        bp = trimblock(bp, 0, iplen);
    r = mkechoreply(bp);
    ipriv->out[EchoReply]++;
    ipoput4(icmp->f, r, 0, MAXTTL, DFLTTOS, nil);
    break;
case Unreachable:
    if(p->code > 5)
        msg = unreachcode[1];
    else
        msg = unreachcode[p->code];

    bp->rp += ICMP_IPSIZE+ICMP_HDRSIZE;
    if(blocklen(bp) < MinAdvise){
        ipriv->stats[LenErrs]++;
        goto raise;
    }
    p = (Icmp *)bp->rp;
    pr = Fsrcvpcolx(icmp->f, p->proto);
    if(pr != nil && pr->advise != nil) {
```

```
            // Protocol dispatch
            (*pr->advise)(pr, bp, msg);
            return;
        }

        bp->rp -= ICMP_IPSIZE+ICMP_HDRSIZE;
        goticmpkt(icmp, bp);
        break;
    case TimeExceed:
        if(p->code == 0){
            snprint(m2, sizeof m2, "ttl exceeded at %V", p->src);

            bp->rp += ICMP_IPSIZE+ICMP_HDRSIZE;
            if(blocklen(bp) < MinAdvise){
                ipriv->stats[LenErrs]++;
                goto raise;
            }
            p = (Icmp *)bp->rp;
            pr = Fsrcvpcolx(icmp->f, p->proto);
            if(pr != nil && pr->advise != nil) {
                // Protocol dispatch
                (*pr->advise)(pr, bp, m2);
                return;
            }
            bp->rp -= ICMP_IPSIZE+ICMP_HDRSIZE;
        }

        goticmpkt(icmp, bp);
        break;
    default:
        goticmpkt(icmp, bp);
        break;
    }
    return;

raise:
    freeblist(bp);
}
```

Uses `CsumErrs-209` 344b, `DFLTTOS` 232b, `EchoReply-188` 343a, `EchoRequest-192` 343a, `Fsrcvpcolx()` 331a, `HlenErrs-211` 344b, `ICMP_HDRSIZE-205` 344a, `ICMP_IPSIZE-204` 344a, `InErrors-207` 344b, `InMsgs-206` 344b, `LenErrs-210` 344b, `Logicmp` 233d, `MAXTTL` 81g, `Maxtype-201` 343a, `MinAdvise-202` 343b, `TimeExceed-193` 343a, `Unreachable-189` 343a, `goticmpkt()` 347c, `icmpnames` 343c, `ipoput4()` 93, `mkechoreply()` 348a, `netlog()` 389a, `ptclcsum()` 90d, and `unreachcode-214` 348b.

⟨*function* `icmpadvise` 350⟩≡ (352)

```
void
icmpadvise(Proto *icmp, Block *bp, char *msg)
{
    Conv    **c, *s;
    Icmp    *p;
    ipaddr   dst;
    ushort  recid;

    p = (Icmp *) bp->rp;
    v4tov6(dst, p->dst);
    recid = nhgets(p->icmpid);

    for(c = icmp->conv; *c; c++) {
        s = *c;
        if(s->lport == recid)
        if(ipcmp(s->raddr, dst) == 0){
```

```
            qhangup(s->rq, msg);
            qhangup(s->wq, msg);
            break;
        }
    }
    freeblist(bp);
}
```

Uses ipcmp 23d and v4tov6() 21c.

⟨*function* icmpstats 351a⟩≡ (352)

```
int
icmpstats(Proto *icmp, char *buf, int len)
{
    Icmppriv *priv;
    char *p, *e;
    int i;

    priv = icmp->priv;
    p = buf;
    e = p+len;
    for(i = 0; i < Nstats; i++)
        p = seprint(p, e, "%s: %lud\n", statnames[i], priv->stats[i]);
    for(i = 0; i <= Maxtype; i++){
        if(icmpnames[i])
            p = seprint(p, e, "%s: %lud %lud\n", icmpnames[i], priv->in[i], priv->out[i]);
        else
            p = seprint(p, e, "%d: %lud %lud\n", i, priv->in[i], priv->out[i]);
    }
    return p - buf;
}
```

Uses Maxtype-201 343a, Nstats-212 344b, icmpnames 343c, and statnames-213 344c.

⟨*function* icmpinit 351b⟩≡ (352)

```
void
icmpinit(Fs *fs)
{
    Proto *icmp;

    icmp = smalloc(sizeof(Proto));
    icmp->priv = smalloc(sizeof(Icmppriv));
    icmp->name = "icmp";
    icmp->connect = icmpconnect;
    icmp->announce = icmpannounce;
    icmp->state = icmpstate;
    icmp->create = icmpcreate;
    icmp->close = icmpclose;
    icmp->rcv = icmpiput;
    icmp->stats = icmpstats;
    icmp->ctl = nil;
    icmp->advise = icmpadvise;
    icmp->gc = nil;
    icmp->ipproto = IP_ICMPPROTO;
    icmp->nc = 128;
    icmp->ptclsize = 0;

    Fsproto(fs, icmp);
}
```

Uses Fsproto() 68a, IP_ICMPPROTO-203 344a, icmpadvise() 350, icmpannounce() 345c, icmpclose() 345d, icmpconnect() 345a, icmpcreate() 344e, icmpiput() 348c, icmpstate() 345b, and icmpstats() 351a.

```
⟨kernel/network/ip/icmp.c 352⟩≡
  #include "u.h"
  #include "../port/lib.h"
  #include "mem.h"
  #include "dat.h"
  #include "fns.h"
  #include "../port/error.h"

  #include "ip.h"

  typedef struct Icmppriv Icmppriv;

  ⟨struct Icmp 342⟩

  ⟨enum _anon_ (kernel/network/ip/icmp.c) 343a⟩

  ⟨enum _anon_ (kernel/network/ip/icmp.c)2 343b⟩

  ⟨global icmpnames 343c⟩

  ⟨enum _anon_ (kernel/network/ip/icmp.c)3 344a⟩

  ⟨enum _anon_ (kernel/network/ip/icmp.c)4 344b⟩

  ⟨global statnames((kernel/network/ip/icmp.c)) 344c⟩

  ⟨struct Icmppriv 344d⟩

  static void icmpkick(void *x, Block*);

  ⟨function icmpcreate 344e⟩

  ⟨function icmpconnect 345a⟩

  ⟨function icmpstate 345b⟩

  ⟨function icmpannounce 345c⟩

  ⟨function icmpclose 345d⟩

  ⟨function icmpkick 345e⟩

  ⟨function icmpttlexceeded 346a⟩

  ⟨function icmpunreachable 346b⟩

  ⟨function icmpnoconv 347a⟩

  ⟨function icmpcantfrag 347b⟩

  ⟨function goticmpkt 347c⟩

  ⟨function mkechoreply 348a⟩

  ⟨global unreachcode 348b⟩

  ⟨function icmpiput 348c⟩

  ⟨function icmpadvise 350⟩
```

⟨*function* icmpstats 351a⟩

⟨*function* icmpinit 351b⟩
Uses Icmppriv 344d.

I.8.4 kernel/network/ip/chandial.c

⟨*enum* _anon_ (kernel/network/ip/chandial.c) 353a⟩≡ (355a)
```
enum
{
    Maxstring=  128,
};
```

⟨*struct* DS*(*(kernel/network/ip/chandial.c)*)* 353b⟩≡ (355a)
```
struct DS
{
    char    buf[Maxstring];         /* dist string */
    char    *netdir;
    char    *proto;
    char    *rem;
    char    *local;             /* other args */
    char    *dir;
    Chan    **ctlp;
};
```
Uses Maxstring-96 353a.

⟨*function* chandial 353c⟩≡ (355a)
```
/*
 *  the dialstring is of the form '[/net/]proto!dest'
 */
Chan*
chandial(char *dest, char *local, char *dir, Chan **ctlp)
{
    DS ds;
    char clone[Maxpath];

    ds.local = local;
    ds.dir = dir;
    ds.ctlp = ctlp;

    _dial_string_parse(dest, &ds);
    if(ds.netdir == 0)
        ds.netdir = "/net";

    /* no connection server, don't translate */
    snprint(clone, sizeof(clone), "%s/%s/clone", ds.netdir, ds.proto);
    return call(clone, ds.rem, &ds);
}
```
Uses Maxpath 232b, _dial_string_parse() 354, and call() 353d.

⟨*function* call*(*(kernel/network/ip/chandial.c)*)* 353d⟩≡ (355a)
```
static Chan*
call(char *clone, char *dest, DS *ds)
{
    int n;
    Chan *dchan, *cchan;
    char name[Maxpath], data[Maxpath], *p;
```

```
    cchan = namec(clone, Aopen, ORDWR, 0);

    /* get directory name */
    if(waserror()){
        cclose(cchan);
        nexterror();
    }
    n = devtab[cchan->type]->read(cchan, name, sizeof(name)-1, 0);
    name[n] = 0;
    for(p = name; *p == ' '; p++)
        ;
    snprint(name, sizeof name, "%lud", strtoul(p, 0, 0));
    p = strrchr(clone, '/');
    *p = 0;
    if(ds->dir)
        snprint(ds->dir, Maxpath, "%s/%s", clone, name);
    snprint(data, sizeof(data), "%s/%s/data", clone, name);

    /* connect */
    if(ds->local)
        snprint(name, sizeof(name), "connect %s %s", dest, ds->local);
    else
        snprint(name, sizeof(name), "connect %s", dest);
    devtab[cchan->type]->write(cchan, name, strlen(name), 0);

    /* open data connection */
    dchan = namec(data, Aopen, ORDWR, 0);
    if(ds->ctlp)
        *ds->ctlp = cchan;
    else
        cclose(cchan);
    poperror();
    return dchan;

}
```

Uses Maxpath 232b.

⟨*function* _dial_string_parse*(*(kernel/network/ip/chandial.c)*)* 354⟩≡ (355a)

```
/*
 *  parse a dial string
 */
static void
_dial_string_parse(char *str, DS *ds)
{
    char *p, *p2;

    strncpy(ds->buf, str, Maxstring);
    ds->buf[Maxstring-1] = 0;

    p = strchr(ds->buf, '!');
    if(p == 0) {
        ds->netdir = 0;
        ds->proto = "net";
        ds->rem = ds->buf;
    } else {
        if(*ds->buf != '/' && *ds->buf != '#'){
            ds->netdir = 0;
            ds->proto = ds->buf;
        } else {
            for(p2 = p; *p2 != '/'; p2--)
```

```
                ;
            *p2++ = 0;
            ds->netdir = ds->buf;
            ds->proto = p2;
        }
        *p = 0;
        ds->rem = p + 1;
    }
}
```
Uses Maxstring-96 353a.

⟨kernel/network/ip/chandial.c 355a⟩≡
```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"
#include    "../ip/ip.h"

typedef struct DS DS;
static Chan*    call(char*, char*, DS*);
static void _dial_string_parse(char*, DS*);

⟨enum _anon_ (kernel/network/ip/chandial.c) 353a⟩

⟨struct DS((kernel/network/ip/chandial.c)) 353b⟩

⟨function chandial 353c⟩

⟨function call((kernel/network/ip/chandial.c)) 353d⟩

⟨function _dial_string_parse((kernel/network/ip/chandial.c)) 354⟩
```
Uses DS 353b.

I.8.5 kernel/network/ip/nullmedium.c

⟨kernel/network/ip/nullmedium.c 355b⟩≡
```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

#include "ip.h"

⟨function nullbind 216b⟩

⟨function nullunbind 216c⟩

⟨function nullbwrite 216d⟩

⟨global nullmedium 216a⟩

⟨function nullmediumlink 216e⟩
```

I.8.6 kernel/network/ip/loopbackmedium.c

⟨kernel/network/ip/loopbackmedium.c 356a⟩≡
```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

#include "ip.h"

typedef struct LB LB;

⟨enum _anon_ (kernel/network/ip/loopbackmedium.c) 217b⟩

⟨struct LB 217d⟩

static void loopbackread(void *a);

⟨function loopbackbind 217e⟩

⟨function loopbackunbind 218a⟩

⟨function loopbackbwrite 218c⟩

⟨function loopbackread 218b⟩

⟨global loopbackmedium 217a⟩

⟨function loopbackmediumlink 217c⟩
```
Uses LB 217d.

I.8.7 kernel/network/ip/ethermedium.c

⟨global ipbroadcast 356b⟩≡ (366c)
```
static ipaddr ipbroadcast = {
    0xff,0xff,0xff,0xff,
    0xff,0xff,0xff,0xff,
    0xff,0xff,0xff,0xff,
    0xff,0xff,0xff,0xff,
};
```

⟨global gbemedium 356c⟩≡ (366c)
```
Medium gbemedium =
{
.name=      "gbe",
.hsize=     14,
.mintu=     60,
.maxtu=     9014,
.maclen=    6,
.bind=      etherbind,
.unbind=    etherunbind,
.bwrite=    etherbwrite,
.addmulti=  etheraddmulti,
.remmulti=  etherremmulti,
.ares=      arpenter,
.areg=      sendgarp,
.pref2addr= etherpref2addr,
```

```
};
```

Uses arpenter() 338, etheraddmulti() 360a, etherbind() 77c, etherpref2addr() 366b, etherremmulti() 360b, etherunbind() 357e, and sendgarp() 362.

⟨*struct* Etherrock 357a⟩≡ (366c)

```
struct Etherrock
{
    Fs  *f;     /* file system we belong to */
    Proc    *arpp;      /* arp process */
    Proc    *read4p;    /* reading process (v4)*/
    Proc    *read6p;    /* reading process (v6)*/
    Chan    *mchan4;    /* Data channel for v4 */
    Chan    *achan;     /* Arp channel */
    Chan    *cchan4;    /* Control channel for v4 */
    Chan    *mchan6;    /* Data channel for v6 */
    Chan    *cchan6;    /* Control channel for v6 */
};
```

⟨*enum* _anon_ (kernel/network/ip/ethermedium.c) 357b⟩≡ (366c)

```
/*
 *  ethernet arp request
 */
enum
{
    ARPREQUEST  = 1,
    ARPREPLY    = 2,
};
```

⟨*struct* Etherarp 357c⟩≡ (366c)

```
struct Etherarp
{
    uchar   d[6];
    uchar   s[6];
    uchar   type[2];
    uchar   hrd[2];
    uchar   pro[2];
    uchar   hln;
    uchar   pln;
    uchar   op[2];
    uchar   sha[6];
    uchar   spa[4];
    uchar   tha[6];
    uchar   tpa[4];
};
```

⟨*global* nbmsg 357d⟩≡ (366c)

```
static char *nbmsg = "nonblocking";
```

Uses nbmsg-184 357d.

⟨*function* etherunbind 357e⟩≡ (366c)

```
/*
 *  called with ifc wlock'd
 */
static void
etherunbind(Ipifc *ifc)
{
    Etherrock *er = ifc->arg;

    if(er->read4p)
        postnote(er->read4p, 1, "unbind", 0);
```

```
    if(er->read6p)
        postnote(er->read6p, 1, "unbind", 0);
    if(er->arpp)
        postnote(er->arpp, 1, "unbind", 0);

    /* wait for readers to die */
    while(er->arpp != 0 || er->read4p != 0 || er->read6p != 0)
        tsleep(&up->sleepr, returnfalse, 0, 300);

    if(er->mchan4 != nil)
        cclose(er->mchan4);
    if(er->achan != nil)
        cclose(er->achan);
    if(er->cchan4 != nil)
        cclose(er->cchan4);
    if(er->mchan6 != nil)
        cclose(er->mchan6);
    if(er->cchan6 != nil)
        cclose(er->cchan6);

    free(er);
}
```

⟨*function* etherbwrite*(*(kernel/network/ip/ethermedium.c)*)* 358⟩≡ (366c)

```
/*
 *  called by ipoput with a single block to write with ifc rlock'd
 */
static void
etherbwrite(Ipifc *ifc, Block *bp, int version, uchar *ip)
{
    Etherhdr *eh;
    Arpent *a;
    uchar mac[6];
    Etherrock *er = ifc->arg;

    /* get mac address of destination */
    a = arpget(er->f->arp, bp, version, ifc, ip, mac);
    if(a){
        /* check for broadcast or multicast */
        bp = multicastarp(er->f, a, ifc->m, mac);
        if(bp==nil){
            switch(version){
            case V4:
                sendarp(ifc, a);
                break;
            case V6:
                resolveaddr6(ifc, a);
                break;
            default:
                panic("etherbwrite: version %d", version);
            }
            return;
        }
    }

    /* make it a single block with space for the ether header */
    bp = padblock(bp, ifc->m->hsize);
    if(bp->next)
        bp = concatblock(bp);
    if(BLEN(bp) < ifc->mintu)
```

```
        bp = adjustblock(bp, ifc->mintu);
    eh = (Etherhdr*)bp->rp;

    /* copy in mac addresses and ether type */
    memmove(eh->s, ifc->mac, sizeof(eh->s));
    memmove(eh->d, mac, sizeof(eh->d));

    switch(version){
    case V4:
        eh->t[0] = 0x08;
        eh->t[1] = 0x00;
        devtab[er->mchan4->type]->bwrite(er->mchan4, bp, 0);
        break;
    case V6:
        eh->t[0] = 0x86;
        eh->t[1] = 0xDD;
        devtab[er->mchan6->type]->bwrite(er->mchan6, bp, 0);
        break;
    default:
        panic("etherbwrite2: version %d", version);
    }
    ifc->out++;
}
```

Uses `V4` 21g, `V6` 21g, `arpget()` 336, `multicastarp()` 365b, `resolveaddr6()` 361, and `sendarp()` 360c.

⟨*function* `etherread4` 359⟩≡ (366c)

```
/*
 *  process to read from the ethernet
 */
static void
etherread4(void *a)
{
    Ipifc *ifc;
    Block *bp;
    Etherrock *er;

    ifc = a;
    er = ifc->arg;
    er->read4p = up;    /* hide identity under a rock for unbind */
    if(waserror()){
        er->read4p = 0;
        pexit("hangup", 1);
    }
    for(;;){
        bp = devtab[er->mchan4->type]->bread(er->mchan4, ifc->maxtu, 0);
        if(!canrlock(ifc)){
            freeb(bp);
            continue;
        }
        if(waserror()){
            runlock(ifc);
            nexterror();
        }
        ifc->in++;
        bp->rp += ifc->m->hsize;
        if(ifc->lifc == nil)
            freeb(bp);
        else
            ipiput4(er->f, ifc, bp);
        runlock(ifc);
```

```
        poperror();
    }
}
```

Uses ipiput4() 98.

⟨*function* etheraddmulti 360a⟩≡ (366c)

```
static void
etheraddmulti(Ipifc *ifc, uchar *a, uchar *)
{
    uchar mac[6];
    char buf[64];
    Etherrock *er = ifc->arg;
    int version;

    version = multicastea(mac, a);
    snprint(buf, sizeof buf, "addmulti %E", mac);
    switch(version){
    case V4:
        devtab[er->cchan4->type]->write(er->cchan4, buf, strlen(buf), 0);
        break;
    case V6:
        devtab[er->cchan6->type]->write(er->cchan6, buf, strlen(buf), 0);
        break;
    default:
        panic("etheraddmulti: version %d", version);
    }
}
```

Uses V4 21g, V6 21g, and multicastea() 365a.

⟨*function* etherremmulti 360b⟩≡ (366c)

```
static void
etherremmulti(Ipifc *ifc, uchar *a, uchar *)
{
    uchar mac[6];
    char buf[64];
    Etherrock *er = ifc->arg;
    int version;

    version = multicastea(mac, a);
    snprint(buf, sizeof buf, "remmulti %E", mac);
    switch(version){
    case V4:
        devtab[er->cchan4->type]->write(er->cchan4, buf, strlen(buf), 0);
        break;
    case V6:
        devtab[er->cchan6->type]->write(er->cchan6, buf, strlen(buf), 0);
        break;
    default:
        panic("etherremmulti: version %d", version);
    }
}
```

Uses V4 21g, V6 21g, and multicastea() 365a.

⟨*function* sendarp 360c⟩≡ (366c)

```
/*
 *  send an ethernet arp
 *  (only v4, v6 uses the neighbor discovery, rfc1970)
 */
static void
```

```
sendarp(Ipifc *ifc, Arpent *a)
{
    int n;
    Block *bp;
    Etherarp *e;
    Etherrock *er = ifc->arg;

    /* don't do anything if it's been less than a second since the last */
    if(NOW - a->ctime < 1000){
        arprelease(er->f->arp, a);
        return;
    }

    /* remove all but the last message */
    while((bp = a->hold) != nil){
        if(bp == a->last)
            break;
        a->hold = bp->list;
        freeblist(bp);
    }

    /* try to keep it around for a second more */
    a->ctime = NOW;
    arprelease(er->f->arp, a);

    n = sizeof(Etherarp);
    if(n < a->type->mintu)
        n = a->type->mintu;
    bp = allocb(n);
    memset(bp->rp, 0, n);
    e = (Etherarp*)bp->rp;
    memmove(e->tpa, a->ip+IPv4off, sizeof(e->tpa));
    ipv4local(ifc, e->spa);
    memmove(e->sha, ifc->mac, sizeof(e->sha));
    memset(e->d, 0xff, sizeof(e->d));       /* ethernet broadcast */
    memmove(e->s, ifc->mac, sizeof(e->s));

    hnputs(e->type, ETARP);
    hnputs(e->hrd, 1);
    hnputs(e->pro, ETIP4);
    e->hln = sizeof(e->sha);
    e->pln = sizeof(e->spa);
    hnputs(e->op, ARPREQUEST);
    bp->wp += n;

    devtab[er->achan->type]->bwrite(er->achan, bp, 0);
}
```

Uses ARPREQUEST-182 357b, ETARP 240f, ETIP4 240f, IPv4off 21a, NOW 234a, arprelease() 337a, and ipv4local() 380.

⟨*function* resolveaddr6 361⟩≡ (366c)

```
static void
resolveaddr6(Ipifc *ifc, Arpent *a)
{
    int sflag;
    Block *bp;
    Etherrock *er = ifc->arg;
    ipaddr ipsrc;

    /* don't do anything if it's been less than a second since the last */
    if(NOW - a->ctime < ReTransTimer){
```

```
        arprelease(er->f->arp, a);
        return;
    }

    /* remove all but the last message */
    while((bp = a->hold) != nil){
        if(bp == a->last)
            break;
        a->hold = bp->list;
        freeblist(bp);
    }

    /* try to keep it around for a second more */
    a->ctime = NOW;
    a->rtime = NOW + ReTransTimer;
    if(a->rxtsrem <= 0) {
        arprelease(er->f->arp, a);
        return;
    }

    a->rxtsrem--;
    arprelease(er->f->arp, a);

    if(sflag = ipv6anylocal(ifc, ipsrc))
        icmpns(er->f, ipsrc, sflag, a->ip, TARG_MULTI, ifc->mac);
}
```

Uses NOW 234a, ReTransTimer 334a, TARG_MULTI 498e, arprelease() 337a, icmpns() 474b, and ipv6anylocal() 512b.

⟨*function* sendgarp 362⟩≡ (366c)

```
/*
 *  send a gratuitous arp to refresh arp caches
 */
static void
sendgarp(Ipifc *ifc, uchar *ip)
{
    int n;
    Block *bp;
    Etherarp *e;
    Etherrock *er = ifc->arg;

    /* don't arp for our initial non address */
    if(ipcmp(ip, IPnoaddr) == 0)
        return;

    n = sizeof(Etherarp);
    if(n < ifc->m->mintu)
        n = ifc->m->mintu;
    bp = allocb(n);
    memset(bp->rp, 0, n);
    e = (Etherarp*)bp->rp;
    memmove(e->tpa, ip+IPv4off, sizeof(e->tpa));
    memmove(e->spa, ip+IPv4off, sizeof(e->spa));
    memmove(e->sha, ifc->mac, sizeof(e->sha));
    memset(e->d, 0xff, sizeof(e->d));       /* ethernet broadcast */
    memmove(e->s, ifc->mac, sizeof(e->s));

    hnputs(e->type, ETARP);
    hnputs(e->hrd, 1);
    hnputs(e->pro, ETIP4);
    e->hln = sizeof(e->sha);
```

```
    e->pln = sizeof(e->spa);
    hnputs(e->op, ARPREQUEST);
    bp->wp += n;

    devtab[er->achan->type]->bwrite(er->achan, bp, 0);
}
```

Uses ARPREQUEST-182 357b, ETARP 240f, ETIP4 240f, IPnoaddr 23b, IPv4off 21a, and ipcmp 23d.

⟨*function* recvarp 363⟩≡ (366c)

```
static void
recvarp(Ipifc *ifc)
{
    int n;
    Block *ebp, *rbp;
    Etherarp *e, *r;
    ipaddr ip;
    static uchar eprinted[4];
    Etherrock *er = ifc->arg;

    ebp = devtab[er->achan->type]->bread(er->achan, ifc->maxtu, 0);
    if(ebp == nil)
        return;

    e = (Etherarp*)ebp->rp;
    switch(nhgets(e->op)) {
    default:
        break;

    case ARPREPLY:
        /* check for machine using my ip address */
        v4tov6(ip, e->spa);
        if(iplocalonifc(ifc, ip) || ipproxyifc(er->f, ifc, ip)){
            if(memcmp(e->sha, ifc->mac, sizeof(e->sha)) != 0){
                print("arprep: 0x%E/0x%E also has ip addr %V\n",
                    e->s, e->sha, e->spa);
                break;
            }
        }

        /* make sure we're not entering broadcast addresses */
        if(ipcmp(ip, ipbroadcast) == 0 ||
            !memcmp(e->sha, etherbroadcast, sizeof(e->sha))){
            print("arprep: 0x%E/0x%E cannot register broadcast address %I\n",
                e->s, e->sha, e->spa);
            break;
        }

        arpenter(er->f, V4, e->spa, e->sha, sizeof(e->sha), 0);
        break;

    case ARPREQUEST:
        /* don't answer arps till we know who we are */
        if(ifc->lifc == 0)
            break;

        /* check for machine using my ip or ether address */
        v4tov6(ip, e->spa);
        if(iplocalonifc(ifc, ip) || ipproxyifc(er->f, ifc, ip)){
            if(memcmp(e->sha, ifc->mac, sizeof(e->sha)) != 0){
                if (memcmp(eprinted, e->spa, sizeof(e->spa))){
```

```
                /* print only once */
                print("arpreq: 0x%E also has ip addr %V\n", e->sha, e->spa);
                memmove(eprinted, e->spa, sizeof(e->spa));
            }
        }
    } else {
        if(memcmp(e->sha, ifc->mac, sizeof(e->sha)) == 0){
            print("arpreq: %V also has ether addr %E\n", e->spa, e->sha);
            break;
        }
    }

    /* refresh what we know about sender */
    arpenter(er->f, V4, e->spa, e->sha, sizeof(e->sha), 1);

    /* answer only requests for our address or systems we're proxying for */
    v4tov6(ip, e->tpa);
    if(!iplocalonifc(ifc, ip))
    if(!ipproxyifc(er->f, ifc, ip))
        break;

    n = sizeof(Etherarp);
    if(n < ifc->mintu)
        n = ifc->mintu;
    rbp = allocb(n);
    r = (Etherarp*)rbp->rp;
    memset(r, 0, sizeof(Etherarp));
    hnputs(r->type, ETARP);
    hnputs(r->hrd, 1);
    hnputs(r->pro, ETIP4);
    r->hln = sizeof(r->sha);
    r->pln = sizeof(r->spa);
    hnputs(r->op, ARPREPLY);
    memmove(r->tha, e->sha, sizeof(r->tha));
    memmove(r->tpa, e->spa, sizeof(r->tpa));
    memmove(r->sha, ifc->mac, sizeof(r->sha));
    memmove(r->spa, e->tpa, sizeof(r->spa));
    memmove(r->d, e->sha, sizeof(r->d));
    memmove(r->s, ifc->mac, sizeof(r->s));
    rbp->wp += n;

    devtab[er->achan->type]->bwrite(er->achan, rbp, 0);
  }
  freeb(ebp);
}
```

Uses ARPREPLY-183 357b, ARPREQUEST-182 357b, ETARP 240f, ETIP4 240f, V4 21g, arpenter() 338, etherbroadcast-181 28a, ipbroadcast-180 356b, ipcmp 23d, iplocalonifc() 381a, ipproxyifc() 381b, and v4tov6() 21c.

⟨*function* recvarpproc 364⟩≡ (366c)

```
static void
recvarpproc(void *v)
{
    Ipifc *ifc = v;
    Etherrock *er = ifc->arg;

    er->arpp = up;
    if(waserror()){
        er->arpp = 0;
        pexit("hangup", 1);
    }
```

```
    for(;;)
        recvarp(ifc);
}
```

Uses recvarp() 363.

⟨*function* multicastea 365a⟩≡ (366c)

```
static int
multicastea(uchar *ea, uchar *ip)
{
    int x;

    switch(x = ipismulticast(ip)){
    case V4:
        ea[0] = 0x01;
        ea[1] = 0x00;
        ea[2] = 0x5e;
        ea[3] = ip[13] & 0x7f;
        ea[4] = ip[14];
        ea[5] = ip[15];
        break;
    case V6:
        ea[0] = 0x33;
        ea[1] = 0x33;
        ea[2] = ip[12];
        ea[3] = ip[13];
        ea[4] = ip[14];
        ea[5] = ip[15];
        break;
    }
    return x;
}
```

Uses V4 21g, V6 21g, and ipismulticast() 381c.

⟨*function* multicastarp 365b⟩≡ (366c)

```
/*
 *  fill in an arp entry for broadcast or multicast
 *  addresses.  Return the first queued packet for the
 *  IP address.
 */
static Block*
multicastarp(Fs *f, Arpent *a, Medium *medium, uchar *mac)
{
    /* is it broadcast? */
    switch(ipforme(f, a->ip)){
    case Runi:
        return nil;
    case Rbcast:
        memset(mac, 0xff, 6);
        return arpresolve(f->arp, a, medium, mac);
    default:
        break;
    }

    /* if multicast, fill in mac */
    switch(multicastea(mac, a->ip)){
    case V4:
    case V6:
        return arpresolve(f->arp, a, medium, mac);
    }
```

```
    /* let arp take care of it */
    return nil;
}
```
Uses Rbcast 38e, Runi 38e, V4 21g, V6 21g, arpresolve() 337b, ipforme() 125e, and multicastea() 365a.

⟨*function* ethermediumlink 366a⟩≡ (366c)
```
void
ethermediumlink(void)
{
    addipmedium(&ethermedium);
    addipmedium(&gbemedium);
}
```
Uses addipmedium() 27c, ethermedium 27f, and gbemedium 356c.

⟨*function* etherpref2addr 366b⟩≡ (366c)
```
static void
etherpref2addr(uchar *pref, uchar *ea)
{
    pref[8] = ea[0] | 0x2;
    pref[9] = ea[1];
    pref[10] = ea[2];
    pref[11] = 0xFF;
    pref[12] = 0xFE;
    pref[13] = ea[3];
    pref[14] = ea[4];
    pref[15] = ea[5];
}
```

⟨kernel/network/ip/ethermedium.c 366c⟩≡
```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

#include "../port/netif.h"
#include "ip.h"
#include "ipv6.h"

typedef struct Etherhdr Etherhdr;
typedef struct  Etherrock Etherrock;
typedef struct Etherarp Etherarp;

⟨struct Etherhdr 86a⟩

⟨global ipbroadcast 356b⟩

⟨global etherbroadcast 28a⟩

static void etherread4(void *a);
static void etherread6(void *a);
static void etherbind(Ipifc *ifc, int argc, char **argv);
static void etherunbind(Ipifc *ifc);
static void etherbwrite(Ipifc *ifc, Block *bp, int version, uchar *ip);
static void etheraddmulti(Ipifc *ifc, uchar *a, uchar *ia);
static void etherremmulti(Ipifc *ifc, uchar *a, uchar *ia);
static Block*   multicastarp(Fs *f, Arpent *a, Medium*, uchar *mac);
static void sendarp(Ipifc *ifc, Arpent *a);
static void sendgarp(Ipifc *ifc, uchar*);
```

```
static int  multicastea(uchar *ea, uchar *ip);
static void recvarpproc(void*);
static void resolveaddr6(Ipifc *ifc, Arpent *a);
static void etherpref2addr(uchar *pref, uchar *ea);


⟨global ethermedium (kernel) 27f⟩

⟨global gbemedium 356c⟩

⟨struct Etherrock 357a⟩

⟨enum _anon_ (kernel/network/ip/ethermedium.c) 357b⟩

⟨struct Etherarp 357c⟩

⟨global nbmsg 357d⟩

⟨function etherbind 77c⟩

⟨function etherunbind 357e⟩

⟨function etherbwrite((kernel/network/ip/ethermedium.c)) 358⟩


⟨function etherread4 359⟩


⟨function etherread6 506d⟩

⟨function etheraddmulti 360a⟩

⟨function etherremmulti 360b⟩

⟨function sendarp 360c⟩

⟨function resolveaddr6 361⟩

⟨function sendgarp 362⟩

⟨function recvarp 363⟩

⟨function recvarpproc 364⟩

⟨function multicastea 365a⟩

⟨function multicastarp 365b⟩

⟨function ethermediumlink 366a⟩


⟨function etherpref2addr 366b⟩
```
Uses `Etherarp` 357c, `Etherhdr` 86a, and `Etherrock` 357a.

I.8.8 `kernel/network/ip/pktmedium.c`

⟨*global* `pktmedium` 367⟩≡ (369b)
```
Medium pktmedium =
{
```

```
    .name=      "pkt",
    .hsize=     14,
    .mintu=     40,
    .maxtu=     4*1024,
    .maclen=    6,
    .bind=      pktbind,
    .unbind=    pktunbind,
    .bwrite=    pktbwrite,
    .pktin=     pktin,
};
```

Uses pktbind() 368a, pktbwrite() 368c, pktin() 368d, and pktunbind() 368b.

⟨*function* pktbind 368a⟩≡ (369b)

```
/*
 *  called to bind an IP ifc to an ethernet device
 *  called with ifc wlock'd
 */
static void
pktbind(Ipifc*, int argc, char **argv)
{
    USED(argc, argv);
}
```

⟨*function* pktunbind 368b⟩≡ (369b)

```
/*
 *  called with ifc wlock'd
 */
static void
pktunbind(Ipifc*)
{
}
```

⟨*function* pktbwrite 368c⟩≡ (369b)

```
/*
 *  called by ipoput with a single packet to write
 */
static void
pktbwrite(Ipifc *ifc, Block *bp, int, uchar*)
{
    /* enqueue onto the conversation's rq */
    bp = concatblock(bp);
    if(ifc->conv->snoopers.ref > 0)
        qpass(ifc->conv->sq, copyblock(bp, BLEN(bp)));
    qpass(ifc->conv->rq, bp);
}
```

⟨*function* pktin 368d⟩≡ (369b)

```
/*
 *  called with ifc rlocked when someone write's to 'data'
 */
static void
pktin(Fs *f, Ipifc *ifc, Block *bp)
{
    if(ifc->lifc == nil)
        freeb(bp);
    else {
        if(ifc->conv->snoopers.ref > 0)
            qpass(ifc->conv->sq, copyblock(bp, BLEN(bp)));
        ipiput4(f, ifc, bp);
    }
```

```
  }
```
Uses ipiput4() 98.

⟨*function* pktmediumlink 369a⟩≡ (369b)
```
  void
  pktmediumlink(void)
  {
      addipmedium(&pktmedium);
  }
```
Uses addipmedium() 27c and pktmedium 367.

⟨kernel/network/ip/pktmedium.c 369b⟩≡
```
  #include "u.h"
  #include "../port/lib.h"
  #include "mem.h"
  #include "dat.h"
  #include "fns.h"
  #include "../port/error.h"

  #include "ip.h"


  static void pktbind(Ipifc*, int, char**);
  static void pktunbind(Ipifc*);
  static void pktbwrite(Ipifc*, Block*, int, uchar*);
  static void pktin(Fs*, Ipifc*, Block*);

  ⟨global pktmedium 367⟩

  ⟨function pktbind 368a⟩

  ⟨function pktunbind 368b⟩

  ⟨function pktbwrite 368c⟩

  ⟨function pktin 368d⟩

  ⟨function pktmediumlink 369a⟩
```

I.8.9 kernel/network/ip/iproute.c

⟨kernel/network/ip/iproute.c 369c⟩≡
```
  #include    "u.h"
  #include    "../port/lib.h"
  #include    "mem.h"
  #include    "dat.h"
  #include    "fns.h"
  #include    "../port/error.h"

  #include    "ip.h"

  static void walkadd(Fs*, Route**, Route*);
  static void addnode(Fs*, Route**, Route*);
  static void calcd(Route*);

  ⟨global v4freelist 115b⟩
  ⟨global v4routegeneration 37d⟩
  ⟨global routelock 115a⟩
```

⟨*global* `v6freelist` 507a⟩
⟨*global* `v6routegeneration` 519b⟩

⟨*function* `freeroute` 116a⟩

⟨*function* `allocroute` 115c⟩

⟨*function* `addqueue` 118e⟩

⟨*function* `lcmp` 519a⟩

⟨*enum* `_anon_ (kernel/network/ip/iproute.c)` 116b⟩

⟨*function* `rangecompare` 116c⟩

⟨*function* `copygate` 118a⟩

⟨*function* `walkadd` 118d⟩

⟨*function* `calcd` 119c⟩

⟨*function* `balancetree` 119b⟩

⟨*function* `addnode` 117⟩

⟨*macro* `V4H` 113a⟩

⟨*function* `v4addroute` 114⟩

⟨*macro* `V6H` 507b⟩
⟨*function* `v6addroute` 507c⟩

⟨*function* `looknode` 131b⟩

⟨*function* `v4delroute` 131a⟩

⟨*function* `v6delroute` 508a⟩

⟨*function* `v4lookup` 112a⟩

⟨*function* `v6lookup` 508b⟩

⟨*function* `routetype` 122a⟩

⟨*global* `rformat` 121a⟩

⟨*function* `convroute` 121c⟩

⟨*function* `sprintroute` 121b⟩

⟨*function* `rr` 123a⟩

⟨*function* `ipwalkroutes` 122b⟩

⟨*function* `routeread` 120d⟩

⟨*function* `delroute` 132b⟩

⟨*function* `routeflush` 132a⟩

⟨*function* iproute 125a⟩

⟨*function* printroute 125b⟩

⟨*function* routewrite 123b⟩

I.8.10 kernel/network/ip/ipaux.c

⟨*global* v6Unspecified 371a⟩≡ (373)

```
/*
 *  well known IPv6 addresses
 */
ipaddr v6Unspecified = {
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```

⟨*global* v6loopback*(*(kernel/network/ip/ipaux.c)*)* 371b⟩≡ (373)

```
ipaddr v6loopback = {
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0x01
};
```

⟨*global* v6linklocal*(*(kernel/network/ip/ipaux.c)*)* 371c⟩≡ (373)

```
ipaddr v6linklocal = {
    0xfe, 0x80, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```

⟨*global* v6llpreflen*(*(kernel/network/ip/ipaux.c)*)* 371d⟩≡ (373)

```
//};
int v6llpreflen = 8;    /* link-local prefix length in bytes */
```

Uses v6llpreflen 371d.

⟨*global* v6allnodesN 371e⟩≡ (373)

```
//int v6mcpreflen = 1;  /* multicast prefix length */

ipaddr v6allnodesN = {
    0xff, 0x01, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0x01
};
```

⟨*global* v6allnodesNmask 371f⟩≡ (373)

```
//};
ipaddr v6allnodesNmask = {
    0xff, 0xff, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```

⟨*global* v6allnodesL 372a⟩≡ (373)
```
//int v6aNpreflen = 2;  /* all nodes (N) prefix */

ipaddr v6allnodesL = {
    0xff, 0x02, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0x01
};
```

⟨*global* v6allnodesLmask 372b⟩≡ (373)
```
//};
ipaddr v6allnodesLmask = {
    0xff, 0xff, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```

⟨*global* v6solicitednode*(*(kernel/network/ip/ipaux.c)*)* 372c⟩≡ (373)
```
//int v6aLpreflen = 2;  /* all nodes (L) prefix */

ipaddr v6solicitednode = {
    0xff, 0x02, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0x01,
    0xff, 0, 0, 0
};
```

⟨*enum* _anon_ (kernel/network/ip/ipaux.c) 372d⟩≡ (373)
```
enum
{
    Isprefix= 16,
};
```

⟨*function* ipv62smcast 372e⟩≡ (373)
```
//#define CLASS(p) ((*(uchar*)(p))>>6)

void
ipv62smcast(uchar *smcast, uchar *a)
{
    assert(IPaddrlen == 16);
    memmove(smcast, v6solicitednode, IPaddrlen);
    smcast[13] = a[13];
    smcast[14] = a[14];
    smcast[15] = a[15];
}
```
Uses IPaddrlen 20c and v6solicitednode 372c.

⟨*function* parsemac 372f⟩≡ (373)
```
/*
 *  parse a hex mac address
 */
int
parsemac(uchar *to, char *from, int len)
{
    char nip[4];
    char *p;
    int i;
```

```
    p = from;
    memset(to, 0, len);
    for(i = 0; i < len; i++){
        if(p[0] == '\0' || p[1] == '\0')
            break;

        nip[0] = p[0];
        nip[1] = p[1];
        nip[2] = '\0';
        p += 2;

        to[i] = strtoul(nip, 0, 16);
        if(*p == ':')
            p++;
    }
    return i;
}
```

⟨kernel/network/ip/ipaux.c 373⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"
#include    "ip.h"
#include    "ipv6.h"

//char *v6hdrtypes[Maxhdrtype] =
//{
//  [HBH]       "HopbyHop",
//  [ICMP]      "ICMP",
//  [IGMP]      "IGMP",
//  [GGP]       "GGP",
//  [IPINIP]    "IP",
//  [ST]        "ST",
//  [TCP]       "TCP",
//  [UDP]       "UDP",
//  [ISO_TP4]   "ISO_TP4",
//  [RH]        "Routinghdr",
//  [FH]        "Fraghdr",
//  [IDRP]      "IDRP",
//  [RSVP]      "RSVP",
//  [AH]        "Authhdr",
//  [ESP]       "ESP",
//  [ICMPv6]    "ICMPv6",
//  [NNH]       "Nonexthdr",
//  [ISO_IP]    "ISO_IP",
//  [IGRP]      "IGRP",
//  [OSPF]      "OSPF",
//};
```

⟨*global* v6Unspecified 371a⟩
⟨*global* v6loopback*(*(kernel/network/ip/ipaux.c)*)* 371b⟩

⟨*global* v6linklocal*(*(kernel/network/ip/ipaux.c)*)* 371c⟩

```
//ipaddr v6linklocalmask = {
//  0xff, 0xff, 0xff, 0xff,
//  0xff, 0xff, 0xff, 0xff,
//  0, 0, 0, 0,
```

```
//  0, 0, 0, 0
⟨global v6llpreflen((kernel/network/ip/ipaux.c)) 371d⟩

//ipaddr v6multicast = {
//  0xff, 0, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0
//};
//ipaddr v6multicastmask = {
//  0xff, 0, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0
//};
⟨global v6allnodesN 371e⟩
//ipaddr v6allroutersN = {
//  0xff, 0x01, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0x02
⟨global v6allnodesNmask 371f⟩
⟨global v6allnodesL 372a⟩
//ipaddr v6allroutersL = {
//  0xff, 0x02, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0,
//  0, 0, 0, 0x02
⟨global v6allnodesLmask 372b⟩
⟨global v6solicitednode((kernel/network/ip/ipaux.c)) 372c⟩
//ipaddr v6solicitednodemask = {
//  0xff, 0xff, 0xff, 0xff,
//  0xff, 0xff, 0xff, 0xff,
//  0xff, 0xff, 0xff, 0xff,
//  0xff, 0x0, 0x0, 0x0
//};
//int v6snpreflen = 13;

⟨function ptclcsum 90d⟩

⟨enum _anon_ (kernel/network/ip/ipaux.c) 372d⟩

⟨function ipv62smcast 372e⟩


⟨function parsemac 372f⟩

⟨function iphash 136c⟩

⟨function iphtadd 136d⟩

⟨function iphtrem 137a⟩

⟨function iphtlook 143⟩
```

I.8.11 kernel/network/ip/ipifc.c

⟨*enum* _anon_ (kernel/network/ip/ipifc.c) 374⟩≡ (384b)

```
enum {
```

```
    ⟨constant Maxmedia 27b⟩
    Nself       = Maxmedia*5,
    NHASH       = 1<<6,
    NCACHE      = 256,
    QMAX        = 192*1024-1,
};
```

Uses Maxmedia-98 27b.

⟨*global* sfixedformat 375a⟩≡ (384b)

```
char sfixedformat[] = "device %s maxtu %d sendra %d recvra %d mflag %d oflag"
" %d maxraint %d minraint %d linkmtu %d reachtime %d rxmitra %d ttl %d routerlt"
" %d pktin %lud pktout %lud errin %lud errout %lud\n";
```

Uses sfixedformat 375a.

⟨*global* slineformat 375b⟩≡ (384b)

```
char slineformat[] = "  %-40I %-10M %-40I %-12lud %-12lud\n";
```

Uses slineformat 375b.

⟨*function* ipifcstate 375c⟩≡ (384b)

```
static int
ipifcstate(Conv *c, char *state, int n)
{
    Ipifc *ifc;
    Iplifc *lifc;
    int m;

    ifc = (Ipifc*)c->ptcl;
    m = snprint(state, n, sfixedformat,
        ifc->dev, ifc->maxtu, ifc->sendra6, ifc->recvra6,
        ifc->rp.mflag, ifc->rp.oflag, ifc->rp.maxraint,
        ifc->rp.minraint, ifc->rp.linkmtu, ifc->rp.reachtime,
        ifc->rp.rxmitra, ifc->rp.ttl, ifc->rp.routerlt,
        ifc->in, ifc->out, ifc->inerr, ifc->outerr);

    rlock(ifc);
    for(lifc = ifc->lifc; lifc && n > m; lifc = lifc->next)
        m += snprint(state+m, n - m, slineformat, lifc->local,
            lifc->mask, lifc->remote, lifc->validlt, lifc->preflt);
    if(ifc->lifc == nil)
        m += snprint(state+m, n - m, "\n");
    runlock(ifc);
    return m;
}
```

⟨*function* ipifclocal 375d⟩≡ (384b)

```
static int
ipifclocal(Conv *c, char *state, int n)
{
    Ipifc *ifc;
    Iplifc *lifc;
    Iplink *link;
    int m;

    ifc = (Ipifc*)c->ptcl;
    m = 0;

    rlock(ifc);
    for(lifc = ifc->lifc; lifc; lifc = lifc->next){
        m += snprint(state+m, n - m, "%-40.40I ->", lifc->local);
```

```
        for(link = lifc->link; link; link = link->lifclink)
            m += snprint(state+m, n - m, " %-40.40I", link->self->a);
        m += snprint(state+m, n - m, "\n");
    }
    runlock(ifc);
    return m;
}
```

⟨*function* ipifcinuse 376a⟩≡ (384b)

```
static int
ipifcinuse(Conv *c)
{
    Ipifc *ifc;

    ifc = (Ipifc*)c->ptcl;
    return ifc->m != nil;
}
```

⟨*function* ipifckick 376b⟩≡ (384b)

```
/*
 *  called when a process writes to an interface's 'data'
 */
static void
ipifckick(void *x)
{
    Conv *c = x;
    Block *bp;
    Ipifc *ifc;

    bp = qget(c->wq);
    if(bp == nil)
        return;

    ifc = (Ipifc*)c->ptcl;
    if(!canrlock(ifc)){
        freeb(bp);
        return;
    }
    if(waserror()){
        runlock(ifc);
        nexterror();
    }
    if(ifc->m == nil || ifc->m->pktin == nil)
        freeb(bp);
    else
        (*ifc->m->pktin)(c->p->f, ifc, bp);
    runlock(ifc);
    poperror();
}
```

⟨*function* ipifcconnect 376c⟩≡ (384b)

```
/*
 *  associate an address with the interface.  This wipes out any previous
 *  addresses.  This is a macro that means, remove all the old interfaces
 *  and add a new one.
 */
static char*
ipifcconnect(Conv* c, char **argv, int argc)
{
    char *err;
```

```
    Ipifc *ifc;

    ifc = (Ipifc*)c->ptcl;

    if(ifc->m == nil)
         return "ipifc not yet bound to device";

    if(waserror()){
        wunlock(ifc);
        nexterror();
    }
    wlock(ifc);
    while(ifc->lifc){
        err = ipifcremlifc(ifc, ifc->lifc);
        if(err)
            error(err);
    }
    wunlock(ifc);
    poperror();

    err = ipifcadd(ifc, argv, argc, 0, nil);
    if(err)
        return err;

    Fsconnected(c, nil);
    return nil;
}
```

Uses ipifcadd() 71c.

⟨*global* freeiplink 377a⟩≡ (384b)

```
/*
 *  These structures are unlinked from their chains while
 *  other threads may be using them.  To avoid excessive locking,
 *  just put them aside for a while before freeing them.
 *  called with f->self locked
 */
static Iplink *freeiplink;
```

⟨*global* freeipself 377b⟩≡ (384b)

```
static Ipself *freeipself;
```

⟨*function* iplinkfree 377c⟩≡ (384b)

```
static void
iplinkfree(Iplink *p)
{
    Iplink **l, *np;
    ulong now = NOW;

    l = &freeiplink;
    for(np = *l; np; np = *l){
        if(np->expire > now){
            *l = np->next;
            free(np);
            continue;
        }
        l = &np->next;
    }
    p->expire = now + 5000; /* give other threads 5 secs to get out */
    p->next = nil;
    *l = p;
```

```
}
```
Uses NOW 234a and freeiplink-105 377a.

⟨*function* ipselffree 378a⟩≡ (384b)
```
static void
ipselffree(Ipself *p)
{
    Ipself **l, *np;
    ulong now = NOW;

    l = &freeipself;
    for(np = *l; np; np = *l){
        if(np->expire > now){
            *l = np->next;
            free(np);
            continue;
        }
        l = &np->next;
    }
    p->expire = now + 5000; /* give other threads 5 secs to get out */
    p->next = nil;
    *l = p;
}
```
Uses NOW 234a and freeipself-106 377b.

⟨*global* stformat 378b⟩≡ (384b)
```
static char *stformat = "%-44.44I %2.2d %4.4s\n";
```
Uses stformat-107 378b.

⟨*enum* _anon_ (kernel/network/ip/ipifc.c)2 378c⟩≡ (384b)
```
enum
{
    Nstformat= 41,
};
```

⟨*function* iptentative 378d⟩≡ (384b)
```
int
iptentative(Fs *f, uchar *addr)
{
    Ipself *p;

    p = f->self->hash[hashipa(addr)];
    for(; p; p = p->next){
        if(ipcmp(addr, p->a) == 0)
            return p->link->lifc->tentative;
    }
    return 0;
}
```
Uses hashipa-103 128a and ipcmp 23d.

⟨*function* findprimaryipv4 378e⟩≡ (384b)
```
/*
 *  returns first ip address configured
 */
static void
findprimaryipv4(Fs *f, uchar *local)
{
    Conv **cp, **e;
    Ipifc *ifc;
```

```
    Iplifc *lifc;

    /* find first ifc local address */
    e = &f->ipifc->conv[f->ipifc->nc];
    for(cp = f->ipifc->conv; cp < e; cp++){
        if(*cp == 0)
            continue;
        ifc = (Ipifc*)(*cp)->ptcl;
        if((lifc = ifc->lifc) != nil){
            ipmove(local, lifc->local);
            return;
        }
    }
}
```

Uses `ipmove` 23c.

⟨*function* `findlocalip` 379⟩≡ (384b)

```
/*
 *  find the local address 'closest' to the remote system, copy it to
 *  local and return the ifc for that address
 */
void
findlocalip(Fs *f, uchar *local, uchar *remote)
{
    int version, atype = unspecifiedv6, atypel = unknownv6;
    int atyper, deprecated;
    ipaddr gate, gnet;
    Ipifc *ifc;
    Iplifc *lifc;
    Route *r;

    USED(atype);
    USED(atypel);
    qlock(f->ipifc);
    r = v6lookup(f, remote, nil);
    version = (memcmp(remote, v4prefix, IPv4off) == 0)? V4: V6;

    if(r != nil){
        ifc = r->ifc;
        if(r->type & Rv4)
            v4tov6(gate, r->v4.gate);
        else {
            ipmove(gate, r->v6.gate);
            ipmove(local, v6Unspecified);
        }

        switch(version) {
        case V4:
            /* find ifc address closest to the gateway to use */
            for(lifc = ifc->lifc; lifc; lifc = lifc->next){
                maskip(gate, lifc->mask, gnet);
                if(ipcmp(gnet, lifc->net) == 0){
                    ipmove(local, lifc->local);
                    goto out;
                }
            }
            break;
        case V6:
            /* find ifc address with scope matching the destination */
            atyper = v6addrtype(remote);
```

```
            deprecated = 0;
            for(lifc = ifc->lifc; lifc; lifc = lifc->next){
                atypel = v6addrtype(lifc->local);
                /* prefer appropriate scope */
                if(atypel > atype && atype < atyper ||
                   atypel < atype && atype > atyper){
                    ipmove(local, lifc->local);
                    deprecated = !v6addrcurr(lifc);
                    atype = atypel;
                } else if(atypel == atype){
                    /* avoid deprecated addresses */
                    if(deprecated && v6addrcurr(lifc)){
                        ipmove(local, lifc->local);
                        atype = atypel;
                        deprecated = 0;
                    }
                }
                if(atype == atyper && !deprecated)
                    goto out;
            }
            if(atype >= atyper)
                goto out;
            break;
        default:
            panic("findlocalip: version %d", version);
        }
    }

    switch(version){
    case V4:
        findprimaryipv4(f, local);
        break;
    case V6:
        findprimaryipv6(f, local);
        break;
    default:
        panic("findlocalip2: version %d", version);
    }

out:
    qunlock(f->ipifc);
}
```

Uses IPv4off 21a, Rv4 38e, V4 21g, V6 21g, findprimaryipv4() 378e, findprimaryipv6() 511c, ipcmp 23d, ipmove 23c, maskip() 22d, unknownv6-109 510b, unspecifiedv6-110 510b, v4prefix 20e, v4tov6() 21c, v6Unspecified 371a, v6addrcurr-113 511b, v6addrtype() 511a, and v6lookup() 508b.

⟨*function* ipv4local 380⟩≡ (384b)

```
/*
 *  return first v4 address associated with an interface
 */
int
ipv4local(Ipifc *ifc, uchar *addr)
{
    Iplifc *lifc;

    for(lifc = ifc->lifc; lifc; lifc = lifc->next){
        if(isv4(lifc->local)){
            memmove(addr, lifc->local+IPv4off, IPv4addrlen);
            return 1;
        }
```

```
    }
    return 0;
}
```

Uses IPv4addrlen 20a, IPv4off 21a, and isv4() 21b.

⟨*function* iplocalonifc 381a⟩≡ (384b)

```
/*
 *  see if this address is bound to the interface
 */
Iplifc*
iplocalonifc(Ipifc *ifc, uchar *ip)
{
    Iplifc *lifc;

    for(lifc = ifc->lifc; lifc; lifc = lifc->next)
        if(ipcmp(ip, lifc->local) == 0)
            return lifc;
    return nil;
}
```

Uses ipcmp 23d.

⟨*function* ipproxyifc 381b⟩≡ (384b)

```
/*
 *  See if we're proxying for this address on this interface
 */
int
ipproxyifc(Fs *f, Ipifc *ifc, uchar *ip)
{
    Route *r;
    ipaddr net;
    Iplifc *lifc;

    /* see if this is a direct connected pt to pt address */
    r = v6lookup(f, ip, nil);
    if(r == nil || (r->type & (Rifc|Rproxy)) != (Rifc|Rproxy))
        return 0;

    /* see if this is on the right interface */
    for(lifc = ifc->lifc; lifc; lifc = lifc->next){
        maskip(ip, lifc->mask, net);
        if(ipcmp(net, lifc->remote) == 0)
            return 1;
    }
    return 0;
}
```

Uses Rifc 38e, Rproxy 38e, ipcmp 23d, maskip() 22d, and v6lookup() 508b.

⟨*function* ipismulticast 381c⟩≡ (384b)

```
/*
 *  return multicast version if any
 */
int
ipismulticast(uchar *ip)
{
    if(isv4(ip)){
        if(ip[IPv4off] >= 0xe0 && ip[IPv4off] < 0xf0)
            return V4;
    }
    else if(ip[0] == 0xff)
```

```
        return V6;
    return 0;
}
```

Uses IPv4off 21a, V4 21g, V6 21g, and isv4() 21b.

⟨*function* ipifcaddmulti 382a⟩≡ (384b)

```
/*
 *  add a multicast address to an interface, called with c->car locked
 */
void
ipifcaddmulti(Conv *c, uchar *ma, uchar *ia)
{
    Ipifc *ifc;
    Iplifc *lifc;
    Conv **p;
    Ipmulti *multi, **l;
    Fs *f;

    f = c->p->f;

    for(l = &c->multi; *l; l = &(*l)->next)
        if(ipcmp(ma, (*l)->ma) == 0 && ipcmp(ia, (*l)->ia) == 0)
            return;     /* it's already there */

    multi = *l = smalloc(sizeof(*multi));
    ipmove(multi->ma, ma);
    ipmove(multi->ia, ia);
    multi->next = nil;

    for(p = f->ipifc->conv; *p; p++){
        if((*p)->inuse == 0)
            continue;
        ifc = (Ipifc*)(*p)->ptcl;
        if(waserror()){
            wunlock(ifc);
            nexterror();
        }
        wlock(ifc);
        for(lifc = ifc->lifc; lifc; lifc = lifc->next)
            if(ipcmp(ia, lifc->local) == 0)
                addselfcache(f, ifc, lifc, ma, Rmulti);
        wunlock(ifc);
        poperror();
    }
}
```

Uses Rmulti 38e, addselfcache() 127b, ipcmp 23d, and ipmove 23c.

⟨*function* ipifcremmulti 382b⟩≡ (384b)

```
/*
 *  remove a multicast address from an interface, called with c->car locked
 */
void
ipifcremmulti(Conv *c, uchar *ma, uchar *ia)
{
    Ipmulti *multi, **l;
    Iplifc *lifc;
    Conv **p;
    Ipifc *ifc;
    Fs *f;
```

```
    f = c->p->f;

    for(l = &c->multi; *l; l = &(*l)->next)
        if(ipcmp(ma, (*l)->ma) == 0 && ipcmp(ia, (*l)->ia) == 0)
            break;

    multi = *l;
    if(multi == nil)
        return;     /* we don't have it open */

    *l = multi->next;

    for(p = f->ipifc->conv; *p; p++){
        if((*p)->inuse == 0)
            continue;

        ifc = (Ipifc*)(*p)->ptcl;
        if(waserror()){
            wunlock(ifc);
            nexterror();
        }
        wlock(ifc);
        for(lifc = ifc->lifc; lifc; lifc = lifc->next)
            if(ipcmp(ia, lifc->local) == 0)
                remselfcache(f, ifc, lifc, ma);
        wunlock(ifc);
        poperror();
    }

    free(multi);
}
```

Uses `ipcmp` 23d and `remselfcache()` 128b.

⟨*function* `ipifcjoinmulti` 383a⟩≡ (384b)

```
/*
 *  make lifc's join and leave multicast groups
 */
static char*
ipifcjoinmulti(Ipifc *ifc, char **argv, int argc)
{
    USED(ifc, argv, argc);
    return nil;
}
```

⟨*function* `ipifcleavemulti` 383b⟩≡ (384b)

```
static char*
ipifcleavemulti(Ipifc *ifc, char **argv, int argc)
{
    USED(ifc, argv, argc);
    return nil;
}
```

⟨*function* `ipifcregisterproxy` 383c⟩≡ (384b)

```
static void
ipifcregisterproxy(Fs *f, Ipifc *ifc, uchar *ip)
{
    Conv **cp, **e;
    Ipifc *nifc;
    Iplifc *lifc;
    Medium *m;
```

```
    ipaddr net;

    /* register the address on any network that will proxy for us */
    e = &f->ipifc->conv[f->ipifc->nc];

    if(!isv4(ip)) {             /* V6 */
        for(cp = f->ipifc->conv; cp < e; cp++){
            if(*cp == nil || (nifc = (Ipifc*)(*cp)->ptcl) == ifc)
                continue;
            rlock(nifc);
            m = nifc->m;
            if(m == nil || m->addmulti == nil) {
                runlock(nifc);
                continue;
            }
            for(lifc = nifc->lifc; lifc; lifc = lifc->next){
                maskip(ip, lifc->mask, net);
                if(ipcmp(net, lifc->remote) == 0) {
                    /* add solicited-node multicast addr */
                    ipv62smcast(net, ip);
                    addselfcache(f, nifc, lifc, net, Rmulti);
                    arpenter(f, V6, ip, nifc->mac, 6, 0);
                    // (*m->addmulti)(nifc, net, ip);
                    break;
                }
            }
            runlock(nifc);
        }
    }
    else {                  /* V4 */
        for(cp = f->ipifc->conv; cp < e; cp++){
            if(*cp == nil || (nifc = (Ipifc*)(*cp)->ptcl) == ifc)
                continue;
            rlock(nifc);
            m = nifc->m;
            if(m == nil || m->areg == nil){
                runlock(nifc);
                continue;
            }
            for(lifc = nifc->lifc; lifc; lifc = lifc->next){
                maskip(ip, lifc->mask, net);
                if(ipcmp(net, lifc->remote) == 0){
                    (*m->areg)(nifc, ip);
                    break;
                }
            }
            runlock(nifc);
        }
    }
}
```

Uses `Rmulti` 38e, `V6` 21g, `addselfcache()` 127b, `arpenter()` 338, `ipcmp` 23d, `ipv62smcast()` 372e, `isv4()` 21b, and `maskip()` 22d.

⟨*enum* _anon_ (kernel/network/ip/ipifc.c)4 384a⟩≡ (384b)

```
//}

enum {
    Ngates = 3,
};
```

⟨kernel/network/ip/ipifc.c 384b⟩≡

```
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"

#include "ip.h"
#include "ipv6.h"

//#define DPRINT if(0)print

extern char*    ipifcadd6(Ipifc *ifc, char**argv, int argc);
extern char*    ipifcadd(Ipifc *ifc, char **argv, int argc, int tentative, Iplifc *lifcp);
extern char*    ipifcrem(Ipifc *ifc, char **argv, int argc);

⟨enum _anon_ (kernel/network/ip/ipifc.c) 374⟩

⟨global media 27a⟩

⟨struct Ipself 126c⟩

⟨struct Ipselftab 126a⟩

/*
 *  Multicast addresses are chained onto a Chan so that
 *  we can remove them when the Chan is closed.
 */
typedef struct Ipmcast Ipmcast;
⟨struct Ipmcast 180a⟩

⟨macro hashipa 128a⟩

⟨global tifc 73a⟩

static void addselfcache(Fs *f, Ipifc *ifc, Iplifc *lifc, uchar *a, int type);
static void remselfcache(Fs *f, Ipifc *ifc, Iplifc *lifc, uchar *a);
static char*    ipifcjoinmulti(Ipifc *ifc, char **argv, int argc);
static char*    ipifcleavemulti(Ipifc *ifc, char **argv, int argc);
static void ipifcregisterproxy(Fs*, Ipifc*, uchar*);
static char*    ipifcremlifc(Ipifc*, Iplifc*);

⟨function addipmedium 27c⟩

⟨function ipfindmedium 70c⟩

⟨function ipifcbind 69⟩

⟨function ipifcunbind 83b⟩

⟨global sfixedformat 375a⟩

⟨global slineformat 375b⟩

⟨function ipifcstate 375c⟩

⟨function ipifclocal 375d⟩

⟨function ipifcinuse 376a⟩
```

⟨*function* ipifckick 376b⟩

⟨*function* ipifccreate 68d⟩

⟨*function* ipifcclose 84b⟩

⟨*function* ipifcsetmtu 80d⟩

⟨*function* ipifcadd 71c⟩

⟨*function* ipifcremlifc 82c⟩

⟨*function* ipifcrem 82b⟩

⟨*function* ipifcaddroute 133b⟩

⟨*function* ipifcremroute 133c⟩

⟨*function* ipifcconnect 376c⟩

⟨*function* ipifcra6 510a⟩

⟨*function* ipifcctl 71a⟩

⟨*function* ipifcstats 77a⟩

⟨*function* ipifcinit 67⟩

⟨*function* addselfcache 127b⟩

⟨*global* freeiplink 377a⟩
⟨*global* freeipself 377b⟩

⟨*function* iplinkfree 377c⟩

⟨*function* ipselffree 378a⟩

⟨*function* remselfcache 128b⟩

⟨*global* stformat 378b⟩
⟨*enum* _anon_ (kernel/network/ip/ipifc.c)2 378c⟩

⟨*function* ipselftabread 130d⟩

⟨*function* iptentative 378d⟩

⟨*function* ipforme 125e⟩

⟨*function* findipifc 113e⟩

⟨*enum* _anon_ (kernel/network/ip/ipifc.c)3 510b⟩

⟨*function* v6addrtype 511a⟩

⟨*macro* v6addrcurr 511b⟩

⟨*function* findprimaryipv6 511c⟩

⟨*function* findprimaryipv4 378e⟩

```
⟨function findlocalip 379⟩

⟨function ipv4local 380⟩

⟨function ipv6local 512a⟩

⟨function ipv6anylocal 512b⟩

⟨function iplocalonifc 381a⟩


⟨function ipproxyifc 381b⟩

⟨function ipismulticast 381c⟩

//int
//ipisbm(uchar *ip)
//{
//  if(isv4(ip)){
//      if(ip[IPv4off] >= 0xe0 && ip[IPv4off] < 0xf0)
//          return V4;
//      else if(ipcmp(ip, IPv4bcast) == 0)
//          return V4;
//  }
//  else if(ip[0] == 0xff)
//      return V6;
//  return 0;
//}


⟨function ipifcaddmulti 382a⟩


⟨function ipifcremmulti 382b⟩

⟨function ipifcjoinmulti 383a⟩

⟨function ipifcleavemulti 383b⟩

⟨function ipifcregisterproxy 383c⟩


/* added for new v6 mesg types */
//static void
//adddefroute6(Fs *f, uchar *gate, int force)
//{
//  Route *r;
//
//  r = v6lookup(f, v6Unspecified, nil);
//  /*
//   * route entries generated by all other means take precedence
//   * over router announcements.
//   */
//  if (r && !force && strcmp(r->tag, "ra") != 0)
//      return;
//
//  v6delroute(f, v6Unspecified, v6Unspecified, 1);
//  v6addroute(f, "ra", v6Unspecified, v6Unspecified, gate, 0);
⟨enum _anon_ (kernel/network/ip/ipifc.c)4 384a⟩
```

⟨*function* ipifcadd6 512c⟩

Uses Ipmcast 180a.

I.8.12 kernel/network/ip/netlog.c

⟨*struct* Netlogflag 388a⟩≡ (389b)
```
typedef struct Netlogflag {
    char*   name;
    int mask;
} Netlogflag;
```
Uses Netlogflag 388a.

⟨*global* flags 388b⟩≡ (389b)
```
static Netlogflag flags[] =
{
    { "ppp",    Logppp, },
    { "ip",     Logip, },
    { "fs",     Logfs, },
    { "tcp",    Logtcp, },
    { "icmp",   Logicmp, },
    { "udp",    Logudp, },
    { "compress",   Logcompress, },
    { "gre",    Loggre, },
    { "tcpwin", Logtcp|Logtcpwin, },
    { "tcprxmt",    Logtcp|Logtcprxmt, },
    { "udpmsg", Logudp|Logudpmsg, },
    { "ipmsg",  Logip|Logipmsg, },
    { "esp",    Logesp, },
    { nil,      0, },
};
```
Uses Logcompress 233d, Logesp 233d, Logfs 233d, Loggre 233d, Logicmp 233d, Logip 233d, Logipmsg 233d, Logppp 233d, Logtcp 233d, Logtcprxmt 233d, Logtcpwin 233d, Logudp 233d, and Logudpmsg 233d.

⟨*global* Ebadnetctl 388c⟩≡ (389b)
```
char Ebadnetctl[] = "too few arguments for netlog control message";
```
Uses Ebadnetctl 388c.

⟨*function* netlogclose 388d⟩≡ (389b)
```
void
netlogclose(Fs *f)
{
    lock(f->alog);
    if(waserror()){
        unlock(f->alog);
        nexterror();
    }
    f->alog->opens--;
    if(f->alog->opens == 0){
        free(f->alog->buf);
        f->alog->buf = nil;
    }
    unlock(f->alog);
    poperror();
}
```

⟨*function* netlog 389a⟩≡ (389b)

```
void
netlog(Fs *f, int mask, char *fmt, ...)
{
    char buf[256], *t, *fp;
    int i, n;
    va_list arg;

    if(!(f->alog->logmask & mask))
        return;

    if(f->alog->opens == 0)
        return;

    va_start(arg, fmt);
    n = vseprint(buf, buf+sizeof(buf), fmt, arg) - buf;
    va_end(arg);

    lock(f->alog);
    i = f->alog->len + n - Nlog;
    if(i > 0){
        f->alog->len -= i;
        f->alog->rptr += i;
        if(f->alog->rptr >= f->alog->end)
            f->alog->rptr = f->alog->buf + (f->alog->rptr - f->alog->end);
    }
    t = f->alog->rptr + f->alog->len;
    fp = buf;
    f->alog->len += n;
    while(n-- > 0){
        if(t >= f->alog->end)
            t = f->alog->buf + (t - f->alog->end);
        *t++ = *fp++;
    }
    unlock(f->alog);

    wakeup(f->alog);
}
```

Uses Nlog-253 189a.

⟨kernel/network/ip/netlog.c 389b⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"
#include    "../ip/ip.h"

⟨enum _anon_ (kernel/network/ip/netlog.c) 189a⟩

⟨struct Netlog 188b⟩

⟨struct Netlogflag 388a⟩

⟨global flags 388b⟩

⟨global Ebadnetctl 388c⟩

⟨enum _anon_ (kernel/network/ip/netlog.c)2 191b⟩
```

⟨*global* `routecmd` 191c⟩

⟨*function* `netloginit` 189c⟩

⟨*function* `netlogopen` 190a⟩

⟨*function* `netlogclose` 388d⟩

⟨*function* `netlogready` 191a⟩

⟨*function* `netlogread` 190b⟩

⟨*function* `netlogctl` 191d⟩

⟨*function* `netlog` 389a⟩

I.8.13 kernel/network/ip/ptclbsum.c

⟨*global* `endian`*(*`(kernel/network/ip/ptclbsum.c)`*)* 390a⟩≡ (391a)

```
static  short   endian  = 1;
```

Uses `endian-185` 390a.

⟨*global* `aendian`*(*`(kernel/network/ip/ptclbsum.c)`*)* 390b⟩≡ (391a)

```
static  uchar*  aendian = (uchar*)&endian;
```

Uses `aendian-186` 390b and `endian-185` 390a.

⟨*constant* `LITTLE`*(*`(kernel/network/ip/ptclbsum.c)`*)* 390c⟩≡ (391a)

```
#define LITTLE  *aendian
```

⟨*function* `ptclbsum`*(*`(kernel/network/ip/ptclbsum.c)`*)* 390d⟩≡ (391a)

```
ushort
ptclbsum(uchar *addr, int len)
{
    ulong losum, hisum, mdsum, x;
    ulong t1, t2;

    losum = 0;
    hisum = 0;
    mdsum = 0;

    x = 0;
    if((ulong)addr & 1) {
        if(len) {
            hisum += addr[0];
            len--;
            addr++;
        }
        x = 1;
    }
    while(len >= 16) {
        t1 = *(ushort*)(addr+0);
        t2 = *(ushort*)(addr+2);    mdsum += t1;
        t1 = *(ushort*)(addr+4);    mdsum += t2;
        t2 = *(ushort*)(addr+6);    mdsum += t1;
        t1 = *(ushort*)(addr+8);    mdsum += t2;
        t2 = *(ushort*)(addr+10);   mdsum += t1;
        t1 = *(ushort*)(addr+12);   mdsum += t2;
        t2 = *(ushort*)(addr+14);   mdsum += t1;
```

```
        mdsum += t2;
        len -= 16;
        addr += 16;
    }
    while(len >= 2) {
        mdsum += *(ushort*)addr;
        len -= 2;
        addr += 2;
    }
    if(x) {
        if(len)
            losum += addr[0];
        if(LITTLE)
            losum += mdsum;
        else
            hisum += mdsum;
    } else {
        if(len)
            hisum += addr[0];
        if(LITTLE)
            hisum += mdsum;
        else
            losum += mdsum;
    }

    losum += hisum >> 8;
    losum += (hisum & 0xff) << 8;
    while(hisum = losum>>16)
        losum = hisum + (losum & 0xffff);

    return losum & 0xffff;
}
```

Uses LITTLE-187 390c.

⟨kernel/network/ip/ptclbsum.c 391a⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"
#include    "ip.h"
```

⟨*global* endian*(*(kernel/network/ip/ptclbsum.c)*)* 390a⟩
⟨*global* aendian*(*(kernel/network/ip/ptclbsum.c)*)* 390b⟩
⟨*constant* LITTLE*(*(kernel/network/ip/ptclbsum.c)*)* 390c⟩

⟨*function* ptclbsum*(*(kernel/network/ip/ptclbsum.c)*)* 390d⟩

I.8.14 kernel/network/ip/il.c

⟨*enum* _anon_ (kernel/network/ip/il.c)3 391b⟩≡ (401b)

```
enum
{
    Seconds  = 1000,
    Iltickms  = 50,  /* time base */
    AckDelay = 2*Iltickms, /* max time twixt message rcvd & ack sent */
    MaxTimeout  = 30*Seconds, /* max time between rexmit */
    QueryTime = 10*Seconds, /* time between subsequent queries */
```

```
    DeathTime = 30*QueryTime,

    MaxRexmit  = 16,  /* max retransmissions before hangup */
    Defaultwin = 20,

    LogAGain = 3,
    AGain  = 1<<LogAGain,
    LogDGain = 2,
    DGain  = 1<<LogDGain,

    DefByteRate = 100,  /* assume a megabit link */
    DefRtt  = 50,  /* cross country on a great day */

    ⟨constant Maxrq(IL) 156a⟩
};
```

Uses `Iltickms-60` 391b, `LogAGain-67` 391b, `LogDGain-69` 391b, `QueryTime-63` 391b, and `Seconds-59` 391b.

⟨*enum* `_anon_ (kernel/network/ip/il.c)4` 392a⟩≡ (401b)

```
enum
{
    Nqt= 8,
};
```

⟨*enum* `_anon_ (kernel/network/ip/il.c)5` 392b⟩≡ (401b)

```
enum
{
    ⟨constant IL_xxxSIZE 155b⟩
    ⟨constant IP_ILPROTO 151a⟩

};
```

⟨*global* `etime` 392c⟩≡ (401b)

```
static char *etime = "connection timed out";
```

Uses `etime-94` 392c.

⟨*function* `ilackq` 392d⟩≡ (401b)

```
void
ilackq(Ilcb *ic, Block *bp)
{
    Block *np;
    int n;

    n = blocklen(bp);

    /* Enqueue a copy on the unacked queue in case this one gets lost */
    np = copyblock(bp, n);
    if(ic->unacked)
        ic->unackedtail->list = np;
    else
        ic->unacked = np;
    ic->unackedtail = np;
    np->list = nil;
    ic->unackedbytes += n;
}
```

⟨*function* `ilrttcalc` 392e⟩≡ (401b)

```
static
void
ilrttcalc(Ilcb *ic, Block *bp)
{
```

```
    int rtt, tt, pt, delay, rate;

    rtt = arch_fastticks(nil) - ic->rttstart;
    rtt = (rtt*scalemul)/scalediv;
    delay = ic->delay;
    rate = ic->rate;

    /* Guard against zero wrap */
    if(rtt > 120000 || rtt < 0)
        return;

    /* this block had to be transmitted after the one acked so count its size */
    ic->rttlen += blocklen(bp)  + IL_IPSIZE + IL_HDRSIZE;

    if(ic->rttlen < 256){
        /* guess fixed delay as rtt of small packets */
        delay += rtt - (delay>>LogAGain);
        if(delay < AGain)
            delay = AGain;
        ic->delay = delay;
    } else {
        /* if packet took longer than avg rtt delay, recalc rate */
        tt = rtt - (delay>>LogAGain);
        if(tt > 0){
            rate += ic->rttlen/tt - (rate>>LogAGain);
            if(rate < AGain)
                rate = AGain;
            ic->rate = rate;
        }
    }

    /* mdev */
    pt = ic->rttlen/(rate>>LogAGain) + (delay>>LogAGain);
    ic->mdev += abs(rtt-pt) - (ic->mdev>>LogDGain);

    if(rtt > ic->maxrtt)
        ic->maxrtt = rtt;
}
```

Uses AGain-68 391b, IL_HDRSIZE-76 155b, IL_IPSIZE-75 155b, LogAGain-67 391b, LogDGain-69 391b, scalediv-92 151c, and scalemul-93 151c.

⟨*function* ilackto 393⟩≡ (401b)

```
void
ilackto(Ilcb *ic, ulong ackto, Block *bp)
{
    Ilhdr *h;
    ulong id;

    if(ic->rttack == ackto)
        ilrttcalc(ic, bp);

    /* Cancel if we've passed the packet we were interested in */
    if(ic->rttack <= ackto)
        ic->rttack = 0;

    qlock(&ic->ackq);
    while(ic->unacked) {
        h = (Ilhdr *)ic->unacked->rp;
        id = nhgetl(h->ilid);
        if(ackto < id)
```

```
            break;

        bp = ic->unacked;
        ic->unacked = bp->list;
        bp->list = nil;
        ic->unackedbytes -= blocklen(bp);
        freeblist(bp);
        ic->rexmit = 0;
        ilsettimeout(ic);
    }
    qunlock(&ic->ackq);
}
```

Uses ilrttcalc() 392e and ilsettimeout() 397b.

⟨*function* ilrexmit 394⟩≡ (401b)

```
void
ilrexmit(Ilcb *ic)
{
    Ilhdr *h;
    Block *nb;
    Conv *c;
//    ulong id; BUG?
    Ilpriv *priv;

    nb = nil;
    qlock(&ic->ackq);
    if(ic->unacked)
        nb = copyblock(ic->unacked, blocklen(ic->unacked));
    qunlock(&ic->ackq);

    if(nb == nil)
        return;

    h = (Ilhdr*)nb->rp;
    h->vihl = IP_VER4;

    h->iltype = Ildataquery;
    hnputl(h->ilack, ic->recvd);
    h->ilspec = ilnextqt(ic);
    h->ilsum[0] = 0;
    h->ilsum[1] = 0;
    hnputs(h->ilsum, ptclcsum(nb, IL_IPSIZE, nhgets(h->illen)));

    c = ic->conv;
    //id = nhgetl(h->ilid);
    //netlog(c->p->f, Logil, "il: rexmit %lud %lud: %d %lud: %I %d/%d\n", id, ic->recvd,
//  ic->rexmit, ic->timeout,
//  c->raddr, c->lport, c->rport);

    ilbackoff(ic);

    ipoput4(c->p->f, nb, 0, c->ttl, c->tos, c);

    /* statistics */
    ic->rxtot++;
    priv = c->p->priv;
    priv->rexmit++;
}
```

Uses IL_IPSIZE-75 155b, IP_VER4 94a, Ildataquery-54 152d, ilbackoff() 398a, ilnextqt() 401a, ipoput4() 93, and ptclcsum() 90d.

⟨*function* ilhangup 395a⟩≡ (401b)

```
void
ilhangup(Conv *s, char *msg)
{
    Ilcb *ic;
    int callout;

    //netlog(s->p->f, Logil, "il: hangup! %I %d/%d: %s\n", s->raddr,
//  s->lport, s->rport, msg?msg:"no reason");

    ic = (Ilcb*)s->ptcl;
    callout = ic->state == Ilsyncer;
    illocalclose(s);

    qhangup(s->rq, msg);
    qhangup(s->wq, msg);

    if(callout)
        Fsconnected(s, msg);
}
```

Uses Ilsyncer-46 152a and illocalclose() 167a.

⟨*function* ilpullup 395b⟩≡ (401b)

```
void
ilpullup(Conv *s)
{
    Ilcb *ic;
    Ilhdr *oh;
    Block *bp;
    ulong oid, dlen;
    Ilpriv *ipriv;

    ic = (Ilcb*)s->ptcl;
    if(ic->state != Ilestablished)
        return;

    qlock(&ic->outo);
    while(ic->outoforder) {
        bp = ic->outoforder;
        oh = (Ilhdr*)bp->rp;
        oid = nhgetl(oh->ilid);
        if(oid <= ic->recvd) {
            ic->outoforder = bp->list;
            freeblist(bp);
            continue;
        }
        if(oid != ic->recvd+1){
            ipriv = s->p->priv;
            ipriv->stats[OutOfOrder]++;
            break;
        }

        ic->recvd = oid;
        ic->outoforder = bp->list;

        bp->list = nil;
        dlen = nhgets(oh->illen)-IL_HDRSIZE;
        bp = trimblock(bp, IL_IPSIZE+IL_HDRSIZE, dlen);
        /*
         * Upper levels don't know about multiple-block
```

```
         * messages so copy all into one (yick).
         */
        bp = concatblock(bp);
        if(bp == 0)
            panic("ilpullup");
        bp = packblock(bp);
        if(bp == 0)
            panic("ilpullup2");
        qpass(s->rq, bp);
    }
    qunlock(&ic->outo);
}
```

Uses IL_HDRSIZE-76 155b, IL_IPSIZE-75 155b, Ilestablished-48 152a, and OutOfOrder-85 154d.

⟨*function* iloutoforder 396⟩≡ (401b)

```
void
iloutoforder(Conv *s, Ilhdr *h, Block *bp)
{
    Ilcb *ic;
    uchar *lid;
    Block *f, **l;
    ulong id, newid;
    Ilpriv *ipriv;

    ipriv = s->p->priv;
    ic = (Ilcb*)s->ptcl;
    bp->list = nil;

    id = nhgetl(h->ilid);
    /* Window checks */
    if(id <= ic->recvd || id > ic->recvd+ic->window) {
        //netlog(s->p->f, Logil, "il: message outside window %lud <%lud-%lud>: %I %d/%d\n",
//   id, ic->recvd, ic->recvd+ic->window, s->raddr, s->lport, s->rport);
        freeblist(bp);
        return;
    }

    /* Packet is acceptable so sort onto receive queue for pullup */
    qlock(&ic->outo);
    if(ic->outoforder == nil)
        ic->outoforder = bp;
    else {
        l = &ic->outoforder;
        for(f = *l; f; f = f->list) {
            lid = ((Ilhdr*)(f->rp))->ilid;
            newid = nhgetl(lid);
            if(id <= newid) {
                if(id == newid) {
                    ipriv->stats[DupMsg]++;
                    ipriv->stats[DupBytes] += blocklen(bp);
                    qunlock(&ic->outo);
                    freeblist(bp);
                    return;
                }
                bp->list = f;
                *l = bp;
                qunlock(&ic->outo);
                return;
            }
            l = &f->list;
```

```
        }
        *l = bp;
    }
    qunlock(&ic->outo);
}
```

Uses DupBytes-88 154d and DupMsg-87 154d.

⟨*function* ilreject 397a⟩≡ (401b)

```
void
ilreject(Fs *f, Ilhdr *inih)
{
    Ilhdr *ih;
    Block *bp;

    bp = allocb(IL_IPSIZE+IL_HDRSIZE);
    bp->wp += IL_IPSIZE+IL_HDRSIZE;

    ih = (Ilhdr *)(bp->rp);
    ih->vihl = IP_VER4;

    /* Ip fields */
    ih->proto = IP_ILPROTO;
    hnputs(ih->illen, IL_HDRSIZE);
    ih->frag[0] = 0;
    ih->frag[1] = 0;
    hnputl(ih->dst, nhgetl(inih->src));
    hnputl(ih->src, nhgetl(inih->dst));
    hnputs(ih->ilsrc, nhgets(inih->ildst));
    hnputs(ih->ildst, nhgets(inih->ilsrc));
    hnputl(ih->ilid, nhgetl(inih->ilack));
    hnputl(ih->ilack, nhgetl(inih->ilid));
    ih->iltype = Ilclose;
    ih->ilspec = 0;
    ih->ilsum[0] = 0;
    ih->ilsum[1] = 0;

    if(ilcksum)
        hnputs(ih->ilsum, ptclcsum(bp, IL_IPSIZE, IL_HDRSIZE));

    ipoput4(f, bp, 0, MAXTTL, DFLTTOS, nil);
}
```

Uses DFLTTOS 232b, IL_HDRSIZE-76 155b, IL_IPSIZE-75 155b, IP_ILPROTO-77 151a, IP_VER4 94a, Ilclose-58 152d, MAXTTL 81g, ilcksum 167b, ipoput4() 93, and ptclcsum() 90d.

⟨*function* ilsettimeout 397b⟩≡ (401b)

```
void
ilsettimeout(Ilcb *ic)
{
    ulong pt;

    pt = (ic->delay>>LogAGain)
        + ic->unackedbytes/(ic->rate>>LogAGain)
        + (ic->mdev>>(LogDGain-1))
        + AckDelay;
    if(pt > MaxTimeout)
        pt = MaxTimeout;
    ic->timeout = NOW + pt;
}
```

Uses AckDelay-61 391b, LogAGain-67 391b, LogDGain-69 391b, MaxTimeout-62 391b, and NOW 234a.

⟨*function* ilbackoff 398a⟩≡ (401b)
```
void
ilbackoff(Ilcb *ic)
{
    ulong pt;
    int i;

    pt = (ic->delay>>LogAGain)
        + ic->unackedbytes/(ic->rate>>LogAGain)
        + (ic->mdev>>(LogDGain-1))
        + AckDelay;
    for(i = 0; i < ic->rexmit; i++)
        pt = pt + (pt>>1);
    if(pt > MaxTimeout)
        pt = MaxTimeout;
    ic->timeout = NOW + pt;

    if(ic->fasttimeout)
        ic->timeout = NOW+Iltickms;

    ic->rexmit++;
}
```
Uses `AckDelay-61` 391b, `Iltickms-60` 391b, `LogAGain-67` 391b, `LogDGain-69` 391b, `MaxTimeout-62` 391b, and `NOW` 234a.

⟨*constant* Tfuture 398b⟩≡ (401b)
```
// complain if two numbers not within an hour of each other
#define Tfuture (1000*60*60)
```

⟨*function* later 398c⟩≡ (401b)
```
int
later(ulong t1, ulong t2, char *x)
{
    int dt;

    dt = t1 - t2;
    if(dt > 0) {
        if(x != nil && dt > Tfuture)
            print("%s: way future %d\n", x, dt);
        return 1;
    }
    if(dt < -Tfuture) {
        if(x != nil)
            print("%s: way past %d\n", x, -dt);
        return 1;
    }
    return 0;
}
```
Uses `Tfuture-95` 398b.

⟨*function* ilackproc 398d⟩≡ (401b)
```
void
ilackproc(void *x)
{
    Ilcb *ic;
    Conv **s, *p;
    Proto *il;

    il = x;

    while(waserror())
```

```
        ;
loop:
    tsleep(&up->sleepr, returnfalse, 0, Iltickms);
    for(s = il->conv; s && *s; s++) {
        p = *s;
        ic = (Ilcb*)p->ptcl;

        switch(ic->state) {
        case Ilclosed:
        case Illistening:
            break;
        case Ilclosing:
            if(later(NOW, ic->timeout, "timeout0")) {
                if(ic->rexmit > MaxRexmit){
                    ilhangup(p, nil);
                    break;
                }
                ilsendctl(p, nil, Ilclose, ic->next, ic->recvd, 0);
                ilbackoff(ic);
            }
            break;

        case Ilsyncee:
        case Ilsyncer:
            if(later(NOW, ic->timeout, "timeout1")) {
                if(ic->rexmit > MaxRexmit){
                    ilhangup(p, etime);
                    break;
                }
                ilsendctl(p, nil, Ilsync, ic->start, ic->recvd, 0);
                ilbackoff(ic);
            }
            break;

        case Ilestablished:
            if(ic->recvd != ic->acksent)
            if(later(NOW, ic->acktime, "acktime"))
                ilsendctl(p, nil, Ilack, ic->next, ic->recvd, 0);

            if(later(NOW, ic->querytime, "querytime")){
                if(later(NOW, ic->lastrecv+DeathTime, "deathtime")){
                    //netlog(il->f, Logil, "il: hangup: deathtime\n");
                    ilhangup(p, etime);
                    break;
                }
                ilsendctl(p, nil, Ilquery, ic->next, ic->recvd, ilnextqt(ic));
                ic->querytime = NOW + QueryTime;
            }

            if(ic->unacked != nil)
            if(later(NOW, ic->timeout, "timeout2")) {
                if(ic->rexmit > MaxRexmit){
                    //netlog(il->f, Logil, "il: hangup: too many rexmits\n");
                    ilhangup(p, etime);
                    break;
                }
                ilsendctl(p, nil, Ilquery, ic->next, ic->recvd, ilnextqt(ic));
                ic->rxquery++;
                ilbackoff(ic);
            }
```

```
            break;
        }
    }
    goto loop;
}
```

⟨*function* ilfreeq 400a⟩≡ (401b)

```
void
ilfreeq(Ilcb *ic)
{
    Block *bp, *next;

    qlock(&ic->ackq);
    for(bp = ic->unacked; bp; bp = next) {
        next = bp->list;
        freeblist(bp);
    }
    ic->unacked = nil;
    qunlock(&ic->ackq);

    qlock(&ic->outo);
    for(bp = ic->outoforder; bp; bp = next) {
        next = bp->list;
        freeblist(bp);
    }
    ic->outoforder = nil;
    qunlock(&ic->outo);
}
```

⟨*function* iladvise 400b⟩≡ (401b)

```
void
iladvise(Proto *il, Block *bp, char *msg)
{
    Ilhdr *h;
    Ilcb *ic;
    ipaddr source, dest;
    ushort psource;
    Conv *s, **p;

    h = (Ilhdr*)(bp->rp);

    v4tov6(dest, h->dst);
    v4tov6(source, h->src);
    psource = nhgets(h->ilsrc);


    /* Look for a connection, unfortunately the destination port is missing */
    qlock(il);
    for(p = il->conv; *p; p++) {
        s = *p;
        if(s->lport == psource)
        if(ipcmp(s->laddr, source) == 0)
        if(ipcmp(s->raddr, dest) == 0){
            qunlock(il);
            ic = (Ilcb*)s->ptcl;
            switch(ic->state){
            case Ilsyncer:
                ilhangup(s, msg);
                break;
            }
```

```
                freeblist(bp);
                return;
            }
        }
        qunlock(il);
        freeblist(bp);
    }
```

Uses Ilsyncer-46 152a, ilhangup() 395a, ipcmp 23d, and v4tov6() 21c.

⟨*function* ilnextqt 401a⟩≡ (401b)

```
    int
    ilnextqt(Ilcb *ic)
    {
        int x;

        qlock(&ic->ackq);
        x = ic->qtx;
        if(++x > Nqt)
            x = 1;
        ic->qtx = x;
        ic->qt[x] = ic->next-1; /* highest xmitted packet */
        ic->qt[0] = ic->qt[x]; /* compatibility with old implementations */
        qunlock(&ic->ackq);

        return x;
    }
```

Uses Nqt-74 392a.

⟨kernel/network/ip/il.c 401b⟩≡

```
    #include "u.h"
    #include "../port/lib.h"
    #include "mem.h"
    #include "dat.h"
    #include "fns.h"
    #include "../port/error.h"

    #include "ip.h"

    // forward decl
    typedef struct Ilcb Ilcb;
    typedef struct Ilhdr Ilhdr;
    typedef struct Ilpriv Ilpriv;

    ⟨enum _anon_ (kernel/network/ip/il.c) 152a⟩

    ⟨global ilstates 152b⟩

    ⟨enum _anon_ (kernel/network/ip/il.c)2 152d⟩

    ⟨global iltype 152e⟩

    ⟨enum _anon_ (kernel/network/ip/il.c)3 391b⟩

    ⟨enum _anon_ (kernel/network/ip/il.c)4 392a⟩

    ⟨struct Ilcb 153⟩

    ⟨enum _anon_ (kernel/network/ip/il.c)5 392b⟩

    ⟨enum mode((kernel/network/ip/il.c)) 156b⟩
```

```
⟨struct Ilhdr 155a⟩

⟨enum _anon_ (kernel/network/ip/il.c)6 154d⟩

⟨global statnames((kernel/network/ip/il.c)) 154e⟩

⟨struct Ilpriv 154a⟩

/* state for query/dataquery messages */

void ilsendctl(Conv*, Ilhdr*, int, ulong, ulong, int);
void ilackq(Ilcb*, Block*);
void ilprocess(Conv*, Ilhdr*, Block*);
void ilpullup(Conv*);
void ilhangup(Conv*, char*);
void ilfreeq(Ilcb*);
void ilrexmit(Ilcb*);
void ilbackoff(Ilcb*);
void ilsettimeout(Ilcb*);
char* ilstart(Conv*, int, int);
void ilackproc(void*);
void iloutoforder(Conv*, Ilhdr*, Block*);
void iliput(Proto*, Ipifc*, Block*);
void iladvise(Proto*, Block*, char*);
int ilnextqt(Ilcb*);
void ilcbinit(Ilcb*);
int later(ulong, ulong, char*);
void ilreject(Fs*, Ilhdr*);
void illocalclose(Conv *c);


⟨global ilcksum 167b⟩
⟨global scalexxx 151c⟩
⟨global etime 392c⟩

⟨function ilconnect 156c⟩

⟨function ilstate 168b⟩

⟨function ilinuse 152c⟩

⟨function ilannounce 156d⟩

⟨function illocalclose 167a⟩

⟨function ilclose 166b⟩

⟨function ilkick 157⟩

⟨function ilcreate 155c⟩

⟨function ilxstats 168a⟩

⟨function ilackq 392d⟩

⟨function ilrttcalc 392e⟩

⟨function ilackto 393⟩
```

```
  ⟨function iliput 158⟩

  ⟨function _ilprocess 162b⟩

  ⟨function ilrexmit 394⟩

  ⟨function ilprocess 162a⟩

  ⟨function ilhangup 395a⟩

  ⟨function ilpullup 395b⟩

  ⟨function iloutoforder 396⟩

  ⟨function ilsendctl 160⟩

  ⟨function ilreject 397a⟩

  ⟨function ilsettimeout 397b⟩

  ⟨function ilbackoff 398a⟩

  ⟨constant Tfuture 398b⟩
  ⟨function later 398c⟩

  ⟨function ilackproc 398d⟩

  ⟨function ilcbinit 166a⟩

  ⟨function ilstart 165⟩

  ⟨function ilfreeq 400a⟩

  ⟨function iladvise 400b⟩

  ⟨function ilnextqt 401a⟩

  ⟨function inittimescale 151b⟩

  ⟨function ilinit 150⟩
```

Uses `Ilcb` 153, `Ilhdr` 155a, and `Ilpriv` 154a.

I.8.15 `kernel/network/ip/udp.c`

⟨*constant* `DPRINT` 403a⟩≡ (405)

```
  #define DPRINT if(0)print
```

⟨*enum* `_anon_ (kernel/network/ip/udp.c)` 403b⟩≡ (405)

```
  enum
  {
      ⟨constant UDP_UDPHDR_SZ 138c⟩

      ⟨constant UDP4_PHDR_OFF 138d⟩
      ⟨constant UDP4_PHDR_SZ 138e⟩
      ⟨constant UDP4_IPHDR_SZ 138b⟩

      ⟨constant UDP6_xxx 514c⟩
      ⟨constant IP_UDPPROTO 135a⟩
```

```
    ⟨constant UDP_USEAD7 148f⟩

    Udprxms     = 200,
    Udptickms   = 100,
    Udpmaxxmit  = 10,
};
```

⟨*function* udpadvise 404⟩≡ (405)

```
void
udpadvise(Proto *udp, Block *bp, char *msg)
{
    Udp4hdr *h4;
    Udp6hdr *h6;
    ipaddr source, dest;
    ushort psource, pdest;
    Conv *s, **p;
    int version;

    h4 = (Udp4hdr*)(bp->rp);
    version = ((h4->vihl&0xF0)==IP_VER6) ? 6 : 4;

    switch(version) {
    case V4:
        v4tov6(dest, h4->udpdst);
        v4tov6(source, h4->udpsrc);
        psource = nhgets(h4->udpsport);
        pdest = nhgets(h4->udpdport);
        break;
    case V6:
        h6 = (Udp6hdr*)(bp->rp);
        ipmove(dest, h6->udpdst);
        ipmove(source, h6->udpsrc);
        psource = nhgets(h6->udpsport);
        pdest = nhgets(h6->udpdport);
        break;
    default:
        panic("udpadvise: version %d", version);
        return;  /* to avoid a warning */
    }

    /* Look for a connection */
    qlock(udp);
    for(p = udp->conv; *p; p++) {
        s = *p;
        if(s->rport == pdest)
        if(s->lport == psource)
        if(ipcmp(s->raddr, dest) == 0)
        if(ipcmp(s->laddr, source) == 0){
            if(s->ignoreadvice)
                break;
            qlock(s);
            qunlock(udp);
            qhangup(s->rq, msg);
            qhangup(s->wq, msg);
            qunlock(s);
            freeblist(bp);
            return;
        }
    }
    qunlock(udp);
```

```
      freeblist(bp);
  }
```
Uses IP_VER6 232b, V4 21g, V6 21g, ipcmp 23d, ipmove 23c, and v4tov6() 21c.

⟨kernel/network/ip/udp.c 405⟩≡
```
  #include    "u.h"
  #include    "../port/lib.h"
  #include    "mem.h"
  #include    "dat.h"
  #include    "fns.h"
  #include    "../port/error.h"

  #include    "ip.h"
  #include    "ipv6.h"


  ⟨constant DPRINT 403a⟩

  ⟨enum _anon_ (kernel/network/ip/udp.c) 403b⟩

  // forward decl
  typedef struct Udp4hdr Udp4hdr;
  typedef struct Udp6hdr Udp6hdr;
  typedef struct Udpstats Udpstats;
  typedef struct Udppriv Udppriv;
  typedef struct Udpcb Udpcb;

  //void (*etherprofiler)(char *name, int qlen);
  void udpkick(void *x, Block *bp);


  ⟨struct Udp4hdr 138a⟩

  ⟨struct Udp6hdr 513b⟩

  ⟨struct Udpstats 137c⟩

  ⟨struct Udppriv 135c⟩

  /*
   *  protocol specific part of Conv
   */
  ⟨struct Udpcb 148c⟩

  ⟨function udpconnect 139c⟩


  ⟨function udpstate 147c⟩

  ⟨function udpannounce 140a⟩

  ⟨function udpcreate 139a⟩

  ⟨function udpclose 139b⟩

  ⟨function udpkick 140b⟩

  ⟨function udpiput 142⟩

  ⟨function udpctl 148e⟩
```

⟨*function* udpadvise 404⟩

⟨*function* udpstats 147b⟩

⟨*function* udpinit 134⟩

Uses Udp4hdr 138a, Udp6hdr 513b, Udpcb 148c, Udppriv 135c, and Udpstats 137c.

I.8.16 kernel/network/ip/tcp.c

⟨*enum* _anon_ (kernel/network/ip/tcp.c) 406⟩≡ (464)

```
enum
{
    QMAX        = 64*1024-1,
    IP_TCPPROTO = 6,

    TCP4_IPLEN  = 8,
    TCP4_PHDRSIZE   = 12,
    TCP4_HDRSIZE    = 20,
    TCP4_TCBPHDRSZ  = 40,
    TCP4_PKT    = TCP4_IPLEN+TCP4_PHDRSIZE,

    TCP6_IPLEN  = 0,
    TCP6_PHDRSIZE   = 40,
    TCP6_HDRSIZE    = 20,
    TCP6_TCBPHDRSZ  = 60,
    TCP6_PKT    = TCP6_IPLEN+TCP6_PHDRSIZE,

    TcptimerOFF = 0,
    TcptimerON  = 1,
    TcptimerDONE    = 2,
    MAX_TIME    = (1<<20),  /* Forever */
    TCP_ACK     = 50,       /* Timed ack sequence in ms */
    MAXBACKMS   = 9*60*1000,    /* longest backoff time (ms) before hangup */

    URG     = 0x20,     /* Data marked urgent */
    ACK     = 0x10,     /* Acknowledge is valid */
    PSH     = 0x08,     /* Whole data pipe is pushed */
    RST     = 0x04,     /* Reset connection */
    SYN     = 0x02,     /* Pkt. is synchronise */
    FIN     = 0x01,     /* Start close down */

    EOLOPT      = 0,
    NOOPOPT     = 1,
    MSSOPT      = 2,
    MSS_LENGTH  = 4,        /* Maximum segment size */
    WSOPT       = 3,
    WS_LENGTH   = 3,        /* Bits to scale window size by */
    MSL2        = 10,
    MSPTICK     = 50,       /* Milliseconds per timer tick */
    DEF_MSS     = 1460,     /* Default maximum segment */
    DEF_MSS6    = 1280,     /* Default maximum segment (min) for v6 */
    DEF_RTT     = 500,      /* Default round trip */
    DEF_KAT     = 120000,   /* Default time (ms) between keep alives */
    TCP_LISTEN  = 0,        /* Listen connection */
    TCP_CONNECT = 1,        /* Outgoing connection */
    SYNACK_RXTIMER  = 250,      /* ms between SYNACK retransmits */

    TCPREXMTTHRESH  = 3,        /* dupack threshhold for rxt */
```

```
    FORCE       = 1,
    CLONE       = 2,
    RETRAN      = 4,
    ACTIVE      = 8,
    SYNACK      = 16,

    LOGAGAIN    = 3,
    LOGDGAIN    = 2,

    Closed      = 0,        /* Connection states */
    Listen,
    Syn_sent,
    Syn_received,
    Established,
    Finwait1,
    Finwait2,
    Close_wait,
    Closing,
    Last_ack,
    Time_wait,

    Maxlimbo    = 1000,     /* maximum procs waiting for response to SYN ACK */
    NLHT        = 256,      /* hash table size, must be a power of 2 */
    LHTMASK     = NLHT-1,

    /*
     * window is 64kb * 2^n
     * these factors determine the ultimate bandwidth-delay product.
     * 64kb * 2^5 = 2mb, or 2× overkill for 100mbps * 70ms.
     */
    Maxqscale   = 4,        /* maximum queuing scale */
    Defadvscale = 4,        /* default advertisement */
};
```

Uses NLHT-327 406, TCP4_IPLEN-270 406, TCP4_PHDRSIZE-271 406, TCP6_IPLEN-275 406, and TCP6_PHDRSIZE-276 406.

⟨*global* tcpstates 407a⟩≡ (464)

```
/* Must correspond to the enumeration above */
char *tcpstates[] =
{
    "Closed",   "Listen",   "Syn_sent", "Syn_received",
    "Established",  "Finwait1", "Finwait2", "Close_wait",
    "Closing",  "Last_ack",     "Time_wait"
};
```

⟨*struct* Tcptimer 407b⟩≡ (464)

```
struct Tcptimer
{
    Tcptimer    *next;
    Tcptimer    *prev;
    Tcptimer    *readynext;
    int state;
    int start;
    int count;
    void    (*func)(void*);
    void    *arg;
};
```

⟨*struct* Tcp4hdr 407c⟩≡ (464)

```
struct Tcp4hdr
```

```
{
    uchar   vihl;       /* Version and header length */
    uchar   tos;        /* Type of service */
    uchar   length[2];  /* packet length */
    uchar   id[2];      /* Identification */
    uchar   frag[2];    /* Fragment information */
    uchar   Unused;
    uchar   proto;
    uchar   tcplen[2];
    uchar   tcpsrc[4];
    uchar   tcpdst[4];
    uchar   tcpsport[2];
    uchar   tcpdport[2];
    uchar   tcpseq[4];
    uchar   tcpack[4];
    uchar   tcpflag[2];
    uchar   tcpwin[2];
    uchar   tcpcksum[2];
    uchar   tcpurg[2];
    /* Options segment */
    uchar   tcpopt[1];
};
```

⟨*struct* Tcp6hdr 408a⟩≡ (464)

```
struct Tcp6hdr
{
    uchar   vcf[4];
    uchar   ploadlen[2];
    uchar   proto;
    uchar   ttl;
    ipaddr   tcpsrc;
    ipaddr   tcpdst;
    uchar   tcpsport[2];
    uchar   tcpdport[2];
    uchar   tcpseq[4];
    uchar   tcpack[4];
    uchar   tcpflag[2];
    uchar   tcpwin[2];
    uchar   tcpcksum[2];
    uchar   tcpurg[2];
    /* Options segment */
    uchar   tcpopt[1];
};
```

⟨*struct* Tcp 408b⟩≡ (464)

```
struct  Tcp
{
    ushort  source;
    ushort  dest;
    ulong   seq;
    ulong   ack;
    uchar   flags;
    uchar   update;
    ushort  ws; /* window scale option */
    ulong   wnd;    /* prescaled window*/
    ushort  urg;
    ushort  mss;    /* max segment size option (if not zero) */
    ushort  len;    /* size of data */
};
```

⟨*struct* Reseq 409a⟩≡ (464)

```
struct Reseq
{
    Reseq   *next;
    Tcp seg;
    Block   *bp;
    ushort  length;
};
```

⟨*struct* Tcpctl 409b⟩≡ (464)

```
struct Tcpctl
{
    uchar   state;          /* Connection state */
    uchar   type;           /* Listening or active connection */
    uchar   code;           /* Icmp code */
    struct {
        ulong   una;        /* Unacked data pointer */
        ulong   nxt;        /* Next sequence expected */
        ulong   ptr;        /* Data pointer */
        ulong   wnd;        /* Tcp send window */
        ulong   urg;        /* Urgent data pointer */
        ulong   wl2;
        uint    scale;      /* how much to right shift window */
                    /* in xmitted packets */
        /* to implement tahoe and reno TCP */
        ulong   dupacks;    /* number of duplicate acks rcvd */
        ulong   partialack;
        int recovery;   /* loss recovery flag */
        int retransmit; /* retransmit 1 packet @ una flag */
        int rto;
        ulong   rxt;        /* right window marker for recovery */
                    /* "recover" rfc3782 */
    } snd;
    struct {
        ulong   nxt;        /* Receive pointer to next uchar slot */
        ulong   wnd;        /* Receive window incoming */
        ulong   wsnt;       /* Last wptr sent.  important to */
                    /* track for large bdp */
        ulong   wptr;
        ulong   urg;        /* Urgent pointer */
        ulong   ackptr;     /* last acked sequence */
        int blocked;
        uint    scale;      /* how much to left shift window in */
                    /* rcv'd packets */
    } rcv;
    ulong   iss;            /* Initial sequence number */
    ulong   cwind;          /* Congestion window */
    ulong   abcbytes;       /* appropriate byte counting rfc 3465 */
    uint    scale;          /* desired snd.scale */
    ulong   ssthresh;       /* Slow start threshold */
    int resent;         /* Bytes just resent */
    int irs;            /* Initial received squence */
    ushort  mss;            /* Maximum segment size */
    int rerecv;         /* Overlap of data rerecevived */
    ulong   window;         /* Our receive window (queue) */
    uint    qscale;         /* Log2 of our receive window (queue) */
    uchar   backoff;        /* Exponential backoff counter */
    int backedoff;      /* ms we've backed off for rexmits */
    uchar   flags;          /* State flags */
    Reseq   *reseq;         /* Resequencing queue */
```

```
    int nreseq;
    int reseqlen;
    Tcptimer    timer;          /* Activity timer */
    Tcptimer    acktimer;       /* Acknowledge timer */
    Tcptimer    rtt_timer;      /* Round trip timer */
    Tcptimer    katimer;        /* keep alive timer */
    ulong   rttseq;         /* Round trip sequence */
    int srtt;           /* Smoothed round trip */
    int mdev;           /* Mean deviation of round trip */
    int kacounter;      /* count down for keep alive */
    uint    sndsyntime;     /* time syn sent */
    ulong   time;           /* time Finwait2 or Syn_received was sent */
    ulong   timeuna;        /* snd.una when time was set */
    int nochecksum;     /* non-zero means don't send checksums */
    int flgcnt;         /* number of flags in the sequence (FIN,SEQ) */

    union {
        Tcp4hdr tcp4hdr;
        Tcp6hdr tcp6hdr;
    } protohdr;     /* prototype header */
};
```

Uses __anon_struct_37 409b, __anon_struct_38 409b, and __anon_struct_39 409b.

⟨*struct* Limbo 410a⟩≡ (464)

```
struct Limbo
{
    Limbo   *next;

    ipaddr   laddr;
    ipaddr   raddr;
    ushort  lport;
    ushort  rport;
    ulong   irs;        /* initial received sequence */
    ulong   iss;        /* initial sent sequence */
    ushort  mss;        /* mss from the other end */
    ushort  rcvscale;   /* how much to scale rcvd windows */
    ushort  sndscale;   /* how much to scale sent windows */
    ulong   lastsend;   /* last time we sent a synack */
    uchar   version;    /* v4 or v6 */
    uchar   rexmits;    /* number of retransmissions */
};
```

⟨*global* tcp_irtt 410b⟩≡ (464)

```
int tcp_irtt = DEF_RTT; /* Initial guess at round trip time */
```

Uses DEF_RTT-302 406 and tcp_irtt 410b.

⟨*enum* _anon_ (kernel/network/ip/tcp.c)2 410c⟩≡ (464)

```
enum {
    /* MIB stats */
    MaxConn,
    Mss,
    ActiveOpens,
    PassiveOpens,
    EstabResets,
    CurrEstab,
    InSegs,
    OutSegs,
    RetransSegs,
    RetransSegsSent,
    RetransTimeouts,
```

```
    InErrs,
    OutRsts,

    /* non-MIB stats */
    CsumErrs,
    HlenErrs,
    LenErrs,
    Resequenced,
    OutOfOrder,
    ReseqBytelim,
    ReseqPktlim,
    Delayack,
    Wopenack,

    Recovery,
    RecoveryDone,
    RecoveryRTO,
    RecoveryNoSeq,
    RecoveryCwind,
    RecoveryPA,

    Nstats
};
```

⟨*global* statnames*(*(kernel/network/ip/tcp.c)*)* 411⟩≡ (464)

```
static char *statnames[Nstats] =
{
[MaxConn]   "MaxConn",
[Mss]       "MaxSegment",
[ActiveOpens]   "ActiveOpens",
[PassiveOpens]  "PassiveOpens",
[EstabResets]   "EstabResets",
[CurrEstab] "CurrEstab",
[InSegs]    "InSegs",
[OutSegs]   "OutSegs",
[RetransSegs]   "RetransSegs",
[RetransSegsSent]   "RetransSegsSent",
[RetransTimeouts]   "RetransTimeouts",
[InErrs]    "InErrs",
[OutRsts]   "OutRsts",
[CsumErrs]  "CsumErrs",
[HlenErrs]  "HlenErrs",
[LenErrs]   "LenErrs",
[OutOfOrder]    "OutOfOrder",
[Resequenced]   "Resequenced",
[ReseqBytelim]  "ReseqBytelim",
[ReseqPktlim]   "ReseqPktlim",
[Delayack]  "Delayack",
[Wopenack]  "Wopenack",

[Recovery]  "Recovery",
[RecoveryDone]  "RecoveryDone",
[RecoveryRTO]   "RecoveryRTO",

[RecoveryNoSeq] "RecoveryNoSeq",
[RecoveryCwind] "RecoveryCwind",
[RecoveryPA]    "RecoveryPA",
};
```

Uses Nstats-359 410c.

⟨*struct* Tcppriv 412a⟩≡ (464)
```
struct Tcppriv
{
    /* List of active timers */
    QLock   tl;
    Tcptimer *timers;

    /* hash table for matching conversations */
    Ipht    ht;

    /* calls in limbo waiting for an ACK to our SYN ACK */
    int nlimbo;
    Limbo   *lht[NLHT];

    /* for keeping track of tcpackproc */
    QLock   apl;
    int ackprocstarted;

    uvlong  stats[Nstats];
};
```
Uses NLHT-327 406 and Nstats-359 410c.

⟨*global* tcpporthogdefense 412b⟩≡ (464)
```
/*
 *  Setting tcpporthogdefense to non-zero enables Dong Lin's
 *  solution to hijacked systems staking out port's as a form
 *  of DoS attack.
 *
 *  To avoid stateless Conv hogs, we pick a sequence number at random.  If
 *  that number gets acked by the other end, we shut down the connection.
 *  Look for tcpporthogdefense in the code.
 */
int tcpporthogdefense = 0;
```
Uses tcpporthogdefense 412b.

⟨*function* tcpsetstate 412c⟩≡ (464)
```
static void
tcpsetstate(Conv *s, uchar newstate)
{
    Tcpctl *tcb;
    uchar oldstate;
    Tcppriv *tpriv;

    tpriv = s->p->priv;

    tcb = (Tcpctl*)s->ptcl;

    oldstate = tcb->state;
    if(oldstate == newstate)
        return;

    if(oldstate == Established)
        tpriv->stats[CurrEstab]--;
    if(newstate == Established)
        tpriv->stats[CurrEstab]++;

    switch(newstate) {
    case Closed:
        qclose(s->rq);
        qclose(s->wq);
```

```
        qclose(s->eq);
        break;

    case Close_wait:        /* Remote closes */
        qhangup(s->rq, nil);
        break;
    }

    tcb->state = newstate;

    if(oldstate == Syn_sent && newstate != Closed)
        Fsconnected(s, nil);
}
```

Uses Close_wait-322 406, Closed-315 406, CurrEstab-336 410c, Established-319 406, and Syn_sent-317 406.

⟨*function* tcpconnect 413a⟩≡ (464)

```
static char*
tcpconnect(Conv *c, char **argv, int argc)
{
    char *e;
    Tcpctl *tcb;

    tcb = (Tcpctl*)(c->ptcl);
    if(tcb->state != Closed)
        return Econinuse;

    e = Fsstdconnect(c, argv, argc);
    if(e != nil)
        return e;
    tcpstart(c, TCP_CONNECT);

    return nil;
}
```

Uses Closed-315 406, Fsstdconnect() 56b, TCP_CONNECT-305 406, and tcpstart() 421.

⟨*function* tcpstate 413b⟩≡ (464)

```
static int
tcpstate(Conv *c, char *state, int n)
{
    Tcpctl *s;

    s = (Tcpctl*)(c->ptcl);

    return snprint(state, n,
        "%s qin %d qout %d rq %d.%d srtt %d mdev %d sst %lud cwin %lud "
        "swin %lud>>%d rwin %lud>>%d qscale %d timer.start %d "
        "timer.count %d rerecv %d katimer.start %d katimer.count %d\n",
        tcpstates[s->state],
        c->rq ? qlen(c->rq) : 0,
        c->wq ? qlen(c->wq) : 0,
        s->nreseq, s->reseqlen,
        s->srtt, s->mdev, s->ssthresh,
        s->cwind, s->snd.wnd, s->rcv.scale, s->rcv.wnd, s->snd.scale,
        s->qscale,
        s->timer.start, s->timer.count, s->rerecv,
        s->katimer.start, s->katimer.count);
}
```

Uses tcpstates 407a.

⟨*function* tcpinuse 414a⟩≡ (464)

```
static int
tcpinuse(Conv *c)
{
    Tcpctl *s;

    s = (Tcpctl*)(c->ptcl);
    return s->state != Closed;
}
```

Uses Closed-315 406.

⟨*function* tcpannounce 414b⟩≡ (464)

```
static char*
tcpannounce(Conv *c, char **argv, int argc)
{
    char *e;
    Tcpctl *tcb;

    tcb = (Tcpctl*)(c->ptcl);
    if(tcb->state != Closed)
        return Econinuse;

    e = Fsstdannounce(c, argv, argc);
    if(e != nil)
        return e;
    tcpstart(c, TCP_LISTEN);
    Fsconnected(c, nil);

    return nil;
}
```

Uses Closed-315 406, Fsstdannounce() 58b, TCP_LISTEN-304 406, and tcpstart() 421.

⟨*function* tcpclose 414c⟩≡ (464)

```
/*
 *  tcpclose is always called with the q locked
 */
static void
tcpclose(Conv *c)
{
    Tcpctl *tcb;

    tcb = (Tcpctl*)c->ptcl;

    qhangup(c->rq, nil);
    qhangup(c->wq, nil);
    qhangup(c->eq, nil);
    qflush(c->rq);

    switch(tcb->state) {
    case Listen:
        /*
         *  reset any incoming calls to this listener
         */
        Fsconnected(c, "Hangup");

        localclose(c, nil);
        break;
    case Closed:
    case Syn_sent:
        localclose(c, nil);
```

```
        break;
    case Syn_received:
    case Established:
        tcb->flgcnt++;
        tcb->snd.nxt++;
        tcpsetstate(c, Finwait1);
        tcpoutput(c);
        break;
    case Close_wait:
        tcb->flgcnt++;
        tcb->snd.nxt++;
        tcpsetstate(c, Last_ack);
        tcpoutput(c);
        break;
    }
}
```

Uses Close_wait-322 406, Closed-315 406, Established-319 406, Finwait1-320 406, Last_ack-324 406, Listen-316 406, Syn_received-318 406, Syn_sent-317 406, localclose() 419d, tcpoutput() 449, and tcpsetstate() 412c.

⟨*function* tcpkick 415a⟩≡ (464)

```
static void
tcpkick(void *x)
{
    Conv *s = x;
    Tcpctl *tcb;

    tcb = (Tcpctl*)s->ptcl;

    if(waserror()){
        qunlock(s);
        nexterror();
    }
    qlock(s);

    switch(tcb->state) {
    case Syn_sent:
    case Syn_received:
    case Established:
    case Close_wait:
        /*
         * Push data
         */
        tcpoutput(s);
        break;
    default:
        localclose(s, "Hangup");
        break;
    }

    qunlock(s);
    poperror();
}
```

Uses Close_wait-322 406, Established-319 406, Syn_received-318 406, Syn_sent-317 406, localclose() 419d, and tcpoutput() 449.

⟨*function* tcprcvwin 415b⟩≡ (464)

```
static void
tcprcvwin(Conv *s)              /* Call with tcb locked */
{
    int w;
```

```
    Tcpctl *tcb;

    tcb = (Tcpctl*)s->ptcl;
    w = tcb->window - qlen(s->rq);
    if(w < 0)
        w = 0;
    /* RFC 1122 § 4.2.2.17 do not move right edge of window left */
    if(seq_lt(tcb->rcv.nxt + w, tcb->rcv.wptr))
        w = tcb->rcv.wptr - tcb->rcv.nxt;
    if(w != tcb->rcv.wnd)
    if(w>>tcb->rcv.scale == 0 || tcb->window > 4*tcb->mss && w < tcb->mss/4){
        tcb->rcv.blocked = 1;
        netlog(s->p->f, Logtcp, "tcprcvwin: window %lud qlen %d ws %ud lport %d\n",
            tcb->window, qlen(s->rq), tcb->rcv.scale, s->lport);
    }
    tcb->rcv.wnd = w;
    tcb->rcv.wptr = tcb->rcv.nxt + w;
}
```

Uses Logtcp 233d, netlog() 389a, and seq_lt() 437a.

⟨*function* tcpacktimer 416a⟩≡ (464)

```
static void
tcpacktimer(void *v)
{
    Tcpctl *tcb;
    Conv *s;

    s = v;
    tcb = (Tcpctl*)s->ptcl;

    if(waserror()){
        qunlock(s);
        nexterror();
    }
    qlock(s);
    if(tcb->state != Closed){
        tcb->flags |= FORCE;
        tcpoutput(s);
    }
    qunlock(s);
    poperror();
}
```

Uses Closed-315 406, FORCE-308 406, and tcpoutput() 449.

⟨*function* tcpcongestion 416b⟩≡ (464)

```
static void
tcpcongestion(Tcpctl *tcb)
{
    ulong inflight;

    inflight = tcb->snd.nxt - tcb->snd.una;
    if(inflight > tcb->cwind)
        inflight = tcb->cwind;
    tcb->ssthresh = inflight / 2;
    if(tcb->ssthresh < 2*tcb->mss)
        tcb->ssthresh = 2*tcb->mss;
}
```

⟨*enum* _anon_ (kernel/network/ip/tcp.c)3 417a⟩≡ (464)
```
enum {
    L   = 2,    /* aggressive slow start; legal values ∈ (1.0, 2.0) */
};
```

⟨*function* tcpabcincr 417b⟩≡ (464)
```
static void
tcpabcincr(Tcpctl *tcb, uint acked)
{
    uint limit;

    tcb->abcbytes += acked;
    if(tcb->cwind < tcb->ssthresh){
        /* slow start */
        if(tcb->snd.rto)
            limit = tcb->mss;
        else
            limit = L*tcb->mss;
        tcb->cwind += MIN(tcb->abcbytes, limit);
        tcb->abcbytes = 0;
    } else {
        tcb->snd.rto = 0;
        /* avoidance */
        if(tcb->abcbytes >= tcb->cwind){
            tcb->abcbytes -= tcb->cwind;
            tcb->cwind += tcb->mss;
        }
    }
}
```
Uses L-361 417a.

⟨*function* tcpcreate 417c⟩≡ (464)
```
static void
tcpcreate(Conv *c)
{
    c->rq = qopen(QMAX, Qcoalesce, tcpacktimer, c);
    c->wq = qopen(QMAX, Qkick, tcpkick, c);
}
```
Uses QMAX-268 406, tcpacktimer() 416a, and tcpkick() 415a.

⟨*function* timerstate 417d⟩≡ (464)
```
static void
timerstate(Tcppriv *priv, Tcptimer *t, int newstate)
{
    if(newstate != TcptimerON){
        if(t->state == TcptimerON){
            /* unchain */
            if(priv->timers == t){
                priv->timers = t->next;
                if(t->prev != nil)
                    panic("timerstate1");
            }
            if(t->next)
                t->next->prev = t->prev;
            if(t->prev)
                t->prev->next = t->next;
            t->next = t->prev = nil;
        }
    } else {
        if(t->state != TcptimerON){
```

```
            /* chain */
            if(t->prev != nil || t->next != nil)
                panic("timerstate2");
            t->prev = nil;
            t->next = priv->timers;
            if(t->next)
                t->next->prev = t;
            priv->timers = t;
        }
    }
    t->state = newstate;
}
```

Uses TcptimerON-281 406.

⟨*function* tcpackproc 418⟩≡ (464)

```
static void
tcpackproc(void *a)
{
    Tcptimer *t, *tp, *timeo;
    Proto *tcp;
    Tcppriv *priv;
    int loop;

    tcp = a;
    priv = tcp->priv;

    for(;;) {
        tsleep(&up->sleepr, returnfalse, 0, MSPTICK);

        qlock(&priv->tl);
        timeo = nil;
        loop = 0;
        for(t = priv->timers; t != nil; t = tp) {
            if(loop++ > 10000)
                panic("tcpackproc1");
            tp = t->next;
            if(t->state == TcptimerON) {
                t->count--;
                if(t->count == 0) {
                    timerstate(priv, t, TcptimerDONE);
                    t->readynext = timeo;
                    timeo = t;
                }
            }
        }
        qunlock(&priv->tl);

        loop = 0;
        for(t = timeo; t != nil; t = t->readynext) {
            if(loop++ > 10000)
                panic("tcpackproc2");
            if(t->state == TcptimerDONE && t->func != nil && !waserror()){
                (*t->func)(t->arg);
                poperror();
            }
        }

        limborexmit(tcp);
    }
}
```

Uses MSPTICK-299 406, TcptimerDONE-282 406, TcptimerON-281 406, limborexmit() 432, and timerstate() 417d.

⟨*function* tcpgo 419a⟩≡ (464)

```
static void
tcpgo(Tcppriv *priv, Tcptimer *t)
{
    if(t == nil || t->start == 0)
        return;

    qlock(&priv->tl);
    t->count = t->start;
    timerstate(priv, t, TcptimerON);
    qunlock(&priv->tl);
}
```

Uses TcptimerON-281 406 and timerstate() 417d.

⟨*function* tcphalt 419b⟩≡ (464)

```
static void
tcphalt(Tcppriv *priv, Tcptimer *t)
{
    if(t == nil)
        return;

    qlock(&priv->tl);
    timerstate(priv, t, TcptimerOFF);
    qunlock(&priv->tl);
}
```

Uses TcptimerOFF-280 406 and timerstate() 417d.

⟨*function* backoff 419c⟩≡ (464)

```
static int
backoff(int n)
{
    return 1 << n;
}
```

⟨*function* localclose 419d⟩≡ (464)

```
static void
localclose(Conv *s, char *reason)   /* called with tcb locked */
{
    Tcpctl *tcb;
    Tcppriv *tpriv;

    tpriv = s->p->priv;
    tcb = (Tcpctl*)s->ptcl;

    iphtrem(&tpriv->ht, s);

    tcphalt(tpriv, &tcb->timer);
    tcphalt(tpriv, &tcb->rtt_timer);
    tcphalt(tpriv, &tcb->acktimer);
    tcphalt(tpriv, &tcb->katimer);

    /* Flush reassembly queue; nothing more can arrive */
    dumpreseq(tcb);

    if(tcb->state == Syn_sent)
        Fsconnected(s, reason);
    if(s->state == Announced)
        wakeup(&s->listenr);
```

```
    qhangup(s->rq, reason);
    qhangup(s->wq, reason);

    tcpsetstate(s, Closed);
}
```

Uses `Announced` 34b, `Closed-315` 406, `Syn_sent-317` 406, `dumpreseq()` 458a, `iphtrem()` 137a, `tcphalt()` 419b, and `tcpsetstate()` 412c.

⟨*function* `tcpmtu` 420a⟩≡ (464)

```
/* mtu (- TCP + IP hdr len) of 1st hop */
static int
tcpmtu(Proto *tcp, uchar *addr, int version, uint *scale)
{
    Ipifc *ifc;
    int mtu;

    ifc = findipifc(tcp->f, addr, 0);
    switch(version){
    default:
    case V4:
        mtu = DEF_MSS;
        if(ifc != nil)
            mtu = ifc->maxtu - ifc->m->hsize - (TCP4_PKT + TCP4_HDRSIZE);
        break;
    case V6:
        mtu = DEF_MSS6;
        if(ifc != nil)
            mtu = ifc->maxtu - ifc->m->hsize - (TCP6_PKT + TCP6_HDRSIZE);
        break;
    }
    /*
     * set the ws.  it doesn't commit us to anything.
     * ws is the ultimate limit to the bandwidth-delay product.
     */
    *scale = Defadvscale;

    return mtu;
}
```

Uses `DEF_MSS-300` 406, `DEF_MSS6-301` 406, `Defadvscale-330` 406, `TCP4_HDRSIZE-272` 406, `TCP4_PKT-274` 406, `TCP6_HDRSIZE-277` 406, `TCP6_PKT-279` 406, `V4` 21g, `V6` 21g, and `findipifc()` 113e.

⟨*function* `inittcpctl` 420b⟩≡ (464)

```
static void
inittcpctl(Conv *s, int mode)
{
    Tcpctl *tcb;
    Tcp4hdr* h4;
    Tcp6hdr* h6;
    Tcppriv *tpriv;
    int mss;

    tcb = (Tcpctl*)s->ptcl;

    memset(tcb, 0, sizeof(Tcpctl));

    tcb->ssthresh = QMAX;           /* reset by tcpsetscale() */
    tcb->srtt = tcp_irtt<<LOGAGAIN;
    tcb->mdev = 0;
```

```
    /* setup timers */
    tcb->timer.start = tcp_irtt / MSPTICK;
    tcb->timer.func = tcptimeout;
    tcb->timer.arg = s;
    tcb->rtt_timer.start = MAX_TIME;
    tcb->acktimer.start = TCP_ACK / MSPTICK;
    tcb->acktimer.func = tcpacktimer;
    tcb->acktimer.arg = s;
    tcb->katimer.start = DEF_KAT / MSPTICK;
    tcb->katimer.func = tcpkeepalive;
    tcb->katimer.arg = s;

    mss = DEF_MSS;

    /* create a prototype(pseudo) header */
    if(mode != TCP_LISTEN){
        if(ipcmp(s->laddr, IPnoaddr) == 0)
            findlocalip(s->p->f, s->laddr, s->raddr);

        switch(s->ipversion){
        case V4:
            h4 = &tcb->protohdr.tcp4hdr;
            memset(h4, 0, sizeof(*h4));
            h4->proto = IP_TCPPROTO;
            hnputs(h4->tcpsport, s->lport);
            hnputs(h4->tcpdport, s->rport);
            v6tov4(h4->tcpsrc, s->laddr);
            v6tov4(h4->tcpdst, s->raddr);
            break;
        case V6:
            h6 = &tcb->protohdr.tcp6hdr;
            memset(h6, 0, sizeof(*h6));
            h6->proto = IP_TCPPROTO;
            hnputs(h6->tcpsport, s->lport);
            hnputs(h6->tcpdport, s->rport);
            ipmove(h6->tcpsrc, s->laddr);
            ipmove(h6->tcpdst, s->raddr);
            mss = DEF_MSS6;
            break;
        default:
            panic("inittcpctl: version %d", s->ipversion);
        }
    }

    tcb->mss = tcb->cwind = mss;
    tcb->abcbytes = 0;
    tpriv = s->p->priv;
    tpriv->stats[Mss] = tcb->mss;

    /* default is no window scaling */
    tcpsetscale(s, tcb, 0, 0);
}
```

Uses `DEF_KAT-303` 406, `DEF_MSS-300` 406, `DEF_MSS6-301` 406, `IP_TCPPROTO-269` 406, `IPnoaddr` 23b, `LOGAGAIN-313` 406, `MAX_TIME-283` 406, `MSPTICK-299` 406, `Mss-332` 410c, `QMAX-268` 406, `TCP_ACK-284` 406, `TCP_LISTEN-304` 406, `V4` 21g, `V6` 21g, `findlocalip()` 379, `ipcmp` 23d, `ipmove` 23c, `tcp_irtt` 410b, `tcpacktimer()` 416a, `tcpkeepalive()` 454b, `tcpsetscale()` 463b, `tcptimeout()` 456, and `v6tov4()` 21d.

⟨*function* `tcpstart` 421⟩≡ (464)

```
/*
 *  called with s qlocked
```

```
 */
static void
tcpstart(Conv *s, int mode)
{
    Tcpctl *tcb;
    Tcppriv *tpriv;
    char kpname[KNAMELEN];

    tpriv = s->p->priv;

    if(tpriv->ackprocstarted == 0){
        qlock(&tpriv->apl);
        if(tpriv->ackprocstarted == 0){
            snprint(kpname, sizeof kpname, "#I%dtcpack", s->p->f->dev);
            kproc(kpname, tcpackproc, s->p);
            tpriv->ackprocstarted = 1;
        }
        qunlock(&tpriv->apl);
    }

    tcb = (Tcpctl*)s->ptcl;

    inittcpctl(s, mode);

    iphtadd(&tpriv->ht, s);
    switch(mode) {
    case TCP_LISTEN:
        tpriv->stats[PassiveOpens]++;
        tcb->flags |= CLONE;
        tcpsetstate(s, Listen);
        break;

    case TCP_CONNECT:
        tpriv->stats[ActiveOpens]++;
        tcb->flags |= ACTIVE;
        tcpsndsyn(s, tcb);
        tcpsetstate(s, Syn_sent);
        tcpoutput(s);
        break;
    }
}
```

Uses `ACTIVE-311` 406, `ActiveOpens-333` 410c, `CLONE-309` 406, `Listen-316` 406, `PassiveOpens-334` 410c, `Syn_sent-317` 406, `TCP_CONNECT-305` 406, `TCP_LISTEN-304` 406, `inittcpctl()` 420b, `iphtadd()` 136d, `tcpackproc()` 418, `tcpoutput()` 449, `tcpsetstate()` 412c, and `tcpsndsyn()` 427.

⟨*function* `htontcp6` 422⟩≡ (464)

```
//}

static Block*
htontcp6(Tcp *tcph, Block *data, Tcp6hdr *ph, Tcpctl *tcb)
{
    int dlen;
    Tcp6hdr *h;
    ushort csum;
    ushort hdrlen, optpad = 0;
    uchar *opt;

    hdrlen = TCP6_HDRSIZE;
    if(tcph->flags & SYN){
        if(tcph->mss)
```

```
        hdrlen += MSS_LENGTH;
    if(tcph->ws)
        hdrlen += WS_LENGTH;
    optpad = hdrlen & 3;
    if(optpad)
        optpad = 4 - optpad;
    hdrlen += optpad;
}

if(data) {
    dlen = blocklen(data);
    data = padblock(data, hdrlen + TCP6_PKT);
    if(data == nil)
        return nil;
}
else {
    dlen = 0;
    data = allocb(hdrlen + TCP6_PKT + 64);  /* the 64 pad is to meet mintu's */
    if(data == nil)
        return nil;
    data->wp += hdrlen + TCP6_PKT;
}

/* copy in pseudo ip header plus port numbers */
h = (Tcp6hdr *)(data->rp);
memmove(h, ph, TCP6_TCBPHDRSZ);

/* compose pseudo tcp header, do cksum calculation */
hnputl(h->vcf, hdrlen + dlen);
h->ploadlen[0] = h->ploadlen[1] = h->proto = 0;
h->ttl = ph->proto;

/* copy in variable bits */
hnputl(h->tcpseq, tcph->seq);
hnputl(h->tcpack, tcph->ack);
hnputs(h->tcpflag, (hdrlen<<10) | tcph->flags);
hnputs(h->tcpwin, tcph->wnd>>(tcb != nil ? tcb->snd.scale : 0));
hnputs(h->tcpurg, tcph->urg);

if(tcph->flags & SYN){
    opt = h->tcpopt;
    if(tcph->mss != 0){
        *opt++ = MSSOPT;
        *opt++ = MSS_LENGTH;
        hnputs(opt, tcph->mss);
        opt += 2;
    }
    if(tcph->ws != 0){
        *opt++ = WSOPT;
        *opt++ = WS_LENGTH;
        *opt++ = tcph->ws;
    }
    while(optpad-- > 0)
        *opt++ = NOOPOPT;
}

if(tcb != nil && tcb->nochecksum){
    h->tcpcksum[0] = h->tcpcksum[1] = 0;
} else {
    csum = ptclcsum(data, TCP6_IPLEN, hdrlen+dlen+TCP6_PHDRSIZE);
```

```
        hnputs(h->tcpcksum, csum);
    }

    /* move from pseudo header back to normal ip header */
    memset(h->vcf, 0, 4);
    h->vcf[0] = IP_VER6;
    hnputs(h->ploadlen, hdrlen+dlen);
    h->proto = ph->proto;

    return data;
}
```

Uses IP_VER6 232b, MSSOPT-294 406, MSS_LENGTH-295 406, NOOPOPT-293 406, SYN-290 406, TCP6_HDRSIZE-277 406, TCP6_IPLEN-275 406, TCP6_PHDRSIZE-276 406, TCP6_PKT-279 406, TCP6_TCBPHDRSZ-278 406, WSOPT-296 406, WS_LENGTH-297 406, and ptclcsum() 90d.

⟨*function* htontcp4 424⟩≡ (464)

```
static Block*
htontcp4(Tcp *tcph, Block *data, Tcp4hdr *ph, Tcpctl *tcb)
{
    int dlen;
    Tcp4hdr *h;
    ushort csum;
    ushort hdrlen, optpad = 0;
    uchar *opt;

    hdrlen = TCP4_HDRSIZE;
    if(tcph->flags & SYN){
        if(tcph->mss)
            hdrlen += MSS_LENGTH;
        if(1)
            hdrlen += WS_LENGTH;
        optpad = hdrlen & 3;
        if(optpad)
            optpad = 4 - optpad;
        hdrlen += optpad;
    }

    if(data) {
        dlen = blocklen(data);
        data = padblock(data, hdrlen + TCP4_PKT);
        if(data == nil)
            return nil;
    }
    else {
        dlen = 0;
        data = allocb(hdrlen + TCP4_PKT + 64);  /* the 64 pad is to meet mintu's */
        if(data == nil)
            return nil;
        data->wp += hdrlen + TCP4_PKT;
    }

    /* copy in pseudo ip header plus port numbers */
    h = (Tcp4hdr *)(data->rp);
    memmove(h, ph, TCP4_TCBPHDRSZ);

    /* copy in variable bits */
    hnputs(h->tcplen, hdrlen + dlen);
    hnputl(h->tcpseq, tcph->seq);
    hnputl(h->tcpack, tcph->ack);
    hnputs(h->tcpflag, (hdrlen<<10) | tcph->flags);
```

```
    hnputs(h->tcpwin, tcph->wnd>>(tcb != nil ? tcb->snd.scale : 0));
    hnputs(h->tcpurg, tcph->urg);

    if(tcph->flags & SYN){
        opt = h->tcpopt;
        if(tcph->mss != 0){
            *opt++ = MSSOPT;
            *opt++ = MSS_LENGTH;
            hnputs(opt, tcph->mss);
            opt += 2;
        }
        /* always offer.  rfc1323 §2.2 */
        if(1){
            *opt++ = WSOPT;
            *opt++ = WS_LENGTH;
            *opt++ = tcph->ws;
        }
        while(optpad-- > 0)
            *opt++ = NOOPOPT;
    }

    if(tcb != nil && tcb->nochecksum){
        h->tcpcksum[0] = h->tcpcksum[1] = 0;
    } else {
        csum = ptclcsum(data, TCP4_IPLEN, hdrlen+dlen+TCP4_PHDRSIZE);
        hnputs(h->tcpcksum, csum);
    }

    return data;
}
```

Uses MSSOPT-294 406, MSS_LENGTH-295 406, NOOPOPT-293 406, SYN-290 406, TCP4_HDRSIZE-272 406, TCP4_IPLEN-270 406, TCP4_PHDRSIZE-271 406, TCP4_PKT-274 406, TCP4_TCBPHDRSZ-273 406, WSOPT-296 406, WS_LENGTH-297 406, and ptclcsum() 90d.

⟨*function* ntohtcp6 425⟩≡ (464)

```
static int
ntohtcp6(Tcp *tcph, Block **bpp)
{
    Tcp6hdr *h;
    uchar *optr;
    ushort hdrlen;
    ushort optlen;
    int n;

    *bpp = pullupblock(*bpp, TCP6_PKT+TCP6_HDRSIZE);
    if(*bpp == nil)
        return -1;

    h = (Tcp6hdr *)((*bpp)->rp);
    tcph->source = nhgets(h->tcpsport);
    tcph->dest = nhgets(h->tcpdport);
    tcph->seq = nhgetl(h->tcpseq);
    tcph->ack = nhgetl(h->tcpack);
    hdrlen = (h->tcpflag[0]>>2) & ~3;
    if(hdrlen < TCP6_HDRSIZE) {
        freeblist(*bpp);
        return -1;
    }

    tcph->flags = h->tcpflag[1];
    tcph->wnd = nhgets(h->tcpwin);
```

```
    tcph->urg = nhgets(h->tcpurg);
    tcph->mss = 0;
    tcph->ws = 0;
    tcph->update = 0;
    tcph->len = nhgets(h->ploadlen) - hdrlen;

    *bpp = pullupblock(*bpp, hdrlen+TCP6_PKT);
    if(*bpp == nil)
        return -1;

    optr = h->tcpopt;
    n = hdrlen - TCP6_HDRSIZE;
    while(n > 0 && *optr != EOLOPT) {
        if(*optr == NOOPOPT) {
            n--;
            optr++;
            continue;
        }
        optlen = optr[1];
        if(optlen < 2 || optlen > n)
            break;
        switch(*optr) {
        case MSSOPT:
            if(optlen == MSS_LENGTH)
                tcph->mss = nhgets(optr+2);
            break;
        case WSOPT:
            if(optlen == WS_LENGTH && *(optr+2) <= 14)
                tcph->ws = *(optr+2);
            break;
        }
        n -= optlen;
        optr += optlen;
    }
    return hdrlen;
}
```

Uses EOLOPT-292 406, MSSOPT-294 406, MSS_LENGTH-295 406, NOOPOPT-293 406, TCP6_HDRSIZE-277 406, TCP6_PKT-279 406, WSOPT-296 406, and WS_LENGTH-297 406.

⟨*function* ntohtcp4 426⟩≡ (464)

```
static int
ntohtcp4(Tcp *tcph, Block **bpp)
{
    Tcp4hdr *h;
    uchar *optr;
    ushort hdrlen;
    ushort optlen;
    int n;

    *bpp = pullupblock(*bpp, TCP4_PKT+TCP4_HDRSIZE);
    if(*bpp == nil)
        return -1;

    h = (Tcp4hdr *)((*bpp)->rp);
    tcph->source = nhgets(h->tcpsport);
    tcph->dest = nhgets(h->tcpdport);
    tcph->seq = nhgetl(h->tcpseq);
    tcph->ack = nhgetl(h->tcpack);

    hdrlen = (h->tcpflag[0]>>2) & ~3;
```

```
    if(hdrlen < TCP4_HDRSIZE) {
        freeblist(*bpp);
        return -1;
    }

    tcph->flags = h->tcpflag[1];
    tcph->wnd = nhgets(h->tcpwin);
    tcph->urg = nhgets(h->tcpurg);
    tcph->mss = 0;
    tcph->ws = 0;
    tcph->update = 0;
    tcph->len = nhgets(h->length) - (hdrlen + TCP4_PKT);

    *bpp = pullupblock(*bpp, hdrlen+TCP4_PKT);
    if(*bpp == nil)
        return -1;

    optr = h->tcpopt;
    n = hdrlen - TCP4_HDRSIZE;
    while(n > 0 && *optr != EOLOPT) {
        if(*optr == NOOPOPT) {
            n--;
            optr++;
            continue;
        }
        optlen = optr[1];
        if(optlen < 2 || optlen > n)
            break;
        switch(*optr) {
        case MSSOPT:
            if(optlen == MSS_LENGTH)
                tcph->mss = nhgets(optr+2);
            break;
        case WSOPT:
            if(optlen == WS_LENGTH && *(optr+2) <= 14)
                tcph->ws = *(optr+2);
            break;
        }
        n -= optlen;
        optr += optlen;
    }
    return hdrlen;
}
```

Uses EOLOPT-292 406, MSSOPT-294 406, MSS_LENGTH-295 406, NOOPOPT-293 406, TCP4_HDRSIZE-272 406, TCP4_PKT-274 406, WSOPT-296 406, and WS_LENGTH-297 406.

⟨*function* tcpsndsyn 427⟩≡ (464)

```
/*
 *  For outgoing calls, generate an initial sequence
 *  number and put a SYN on the send queue
 */
static void
tcpsndsyn(Conv *s, Tcpctl *tcb)
{
    Tcppriv *tpriv;

    tcb->iss = (nrand(1<<16)<<16)|nrand(1<<16);
    tcb->rttseq = tcb->iss;
    tcb->snd.wl2 = tcb->iss;
    tcb->snd.una = tcb->iss;
```

```
    tcb->snd.rxt = tcb->iss;
    tcb->snd.ptr = tcb->rttseq;
    tcb->snd.nxt = tcb->rttseq;
    tcb->flgcnt++;
    tcb->flags |= FORCE;
    tcb->sndsyntime = NOW;

    /* set desired mss and scale */
    tcb->mss = tcpmtu(s->p, s->laddr, s->ipversion, &tcb->scale);
    tpriv = s->p->priv;
    tpriv->stats[Mss] = tcb->mss;
}
```

Uses `FORCE-308` 406, `Mss-332` 410c, `NOW` 234a, and `tcpmtu()` 420a.

⟨*function* `sndrst` 428⟩≡ (464)

```
void
sndrst(Proto *tcp, uchar *source, uchar *dest, ushort length, Tcp *seg, uchar version, char *reason)
{
    Block *hbp;
    uchar rflags;
    Tcppriv *tpriv;
    Tcp4hdr ph4;
    Tcp6hdr ph6;

    netlog(tcp->f, Logtcp, "sndrst: %s\n", reason);

    tpriv = tcp->priv;

    if(seg->flags & RST)
        return;

    /* make pseudo header */
    switch(version) {
    case V4:
        memset(&ph4, 0, sizeof(ph4));
        ph4.vihl = IP_VER4;
        v6tov4(ph4.tcpsrc, dest);
        v6tov4(ph4.tcpdst, source);
        ph4.proto = IP_TCPPROTO;
        hnputs(ph4.tcplen, TCP4_HDRSIZE);
        hnputs(ph4.tcpsport, seg->dest);
        hnputs(ph4.tcpdport, seg->source);
        break;
    case V6:
        memset(&ph6, 0, sizeof(ph6));
        ph6.vcf[0] = IP_VER6;
        ipmove(ph6.tcpsrc, dest);
        ipmove(ph6.tcpdst, source);
        ph6.proto = IP_TCPPROTO;
        hnputs(ph6.ploadlen, TCP6_HDRSIZE);
        hnputs(ph6.tcpsport, seg->dest);
        hnputs(ph6.tcpdport, seg->source);
        break;
    default:
        panic("sndrst: version %d", version);
    }

    tpriv->stats[OutRsts]++;
    rflags = RST;
```

```
    /* convince the other end that this reset is in band */
    if(seg->flags & ACK) {
        seg->seq = seg->ack;
        seg->ack = 0;
    }
    else {
        rflags |= ACK;
        seg->ack = seg->seq;
        seg->seq = 0;
        if(seg->flags & SYN)
            seg->ack++;
        seg->ack += length;
        if(seg->flags & FIN)
            seg->ack++;
    }
    seg->flags = rflags;
    seg->wnd = 0;
    seg->urg = 0;
    seg->mss = 0;
    seg->ws = 0;
    switch(version) {
    case V4:
        hbp = htontcp4(seg, nil, &ph4, nil);
        if(hbp == nil)
            return;
        ipoput4(tcp->f, hbp, 0, MAXTTL, DFLTTOS, nil);
        break;
    case V6:
        hbp = htontcp6(seg, nil, &ph6, nil);
        if(hbp == nil)
            return;
        ipoput6(tcp->f, hbp, 0, MAXTTL, DFLTTOS, nil);
        break;
    default:
        panic("sndrst2: version %d", version);
    }
}
```

Uses `ACK-287` 406, `DFLTTOS` 232b, `FIN-291` 406, `IP_TCPPROTO-269` 406, `IP_VER4` 94a, `IP_VER6` 232b, `Logtcp` 233d, `MAXTTL` 81g, `OutRsts-343` 410c, `RST-289` 406, `SYN-290` 406, `TCP4_HDRSIZE-272` 406, `TCP6_HDRSIZE-277` 406, `V4` 21g, `V6` 21g, `htontcp4()` 424, `htontcp6()` 422, `ipmove` 23c, `ipoput4()` 93, `ipoput6()` 486e, `netlog()` 389a, and `v6tov4()` 21d.

⟨*function* `tcphangup` 429⟩≡ (464)

```
/*
 *  send a reset to the remote side and close the conversation
 *  called with s qlocked
 */
static char*
tcphangup(Conv *s)
{
    Tcp seg;
    Tcpctl *tcb;
    Block *hbp;

    tcb = (Tcpctl*)s->ptcl;
    if(waserror())
        return up->errstr;
    if(ipcmp(s->raddr, IPnoaddr) != 0) {
        if(!waserror()){
            memset(&seg, 0, sizeof seg);
            seg.flags = RST | ACK;
```

```
            seg.ack = tcb->rcv.nxt;
            tcb->rcv.ackptr = seg.ack;
            seg.seq = tcb->snd.ptr;
            seg.wnd = 0;
            seg.urg = 0;
            seg.mss = 0;
            seg.ws = 0;
            switch(s->ipversion) {
            case V4:
                tcb->protohdr.tcp4hdr.vihl = IP_VER4;
                hbp = htontcp4(&seg, nil, &tcb->protohdr.tcp4hdr, tcb);
                ipoput4(s->p->f, hbp, 0, s->ttl, s->tos, s);
                break;
            case V6:
                tcb->protohdr.tcp6hdr.vcf[0] = IP_VER6;
                hbp = htontcp6(&seg, nil, &tcb->protohdr.tcp6hdr, tcb);
                ipoput6(s->p->f, hbp, 0, s->ttl, s->tos, s);
                break;
            default:
                panic("tcphangup: version %d", s->ipversion);
            }
            poperror();
        }
    }
    localclose(s, nil);
    poperror();
    return nil;
}
```

Uses ACK-287 406, IP_VER4 94a, IP_VER6 232b, IPnoaddr 23b, RST-289 406, V4 21g, V6 21g, htontcp4() 424, htontcp6() 422, ipcmp 23d, ipoput4() 93, ipoput6() 486e, and localclose() 419d.

⟨*function* sndsynack 430⟩≡ (464)

```
/*
 *  (re)send a SYN ACK
 */
static int
sndsynack(Proto *tcp, Limbo *lp)
{
    Block *hbp;
    Tcp4hdr ph4;
    Tcp6hdr ph6;
    Tcp seg;
    uint scale;

    /* make pseudo header */
    switch(lp->version) {
    case V4:
        memset(&ph4, 0, sizeof(ph4));
        ph4.vihl = IP_VER4;
        v6tov4(ph4.tcpsrc, lp->laddr);
        v6tov4(ph4.tcpdst, lp->raddr);
        ph4.proto = IP_TCPPROTO;
        hnputs(ph4.tcplen, TCP4_HDRSIZE);
        hnputs(ph4.tcpsport, lp->lport);
        hnputs(ph4.tcpdport, lp->rport);
        break;
    case V6:
        memset(&ph6, 0, sizeof(ph6));
        ph6.vcf[0] = IP_VER6;
        ipmove(ph6.tcpsrc, lp->laddr);
```

```
        ipmove(ph6.tcpdst, lp->raddr);
        ph6.proto = IP_TCPPROTO;
        hnputs(ph6.ploadlen, TCP6_HDRSIZE);
        hnputs(ph6.tcpsport, lp->lport);
        hnputs(ph6.tcpdport, lp->rport);
        break;
    default:
        panic("sndrst: version %d", lp->version);
    }

    memset(&seg, 0, sizeof seg);
    seg.seq = lp->iss;
    seg.ack = lp->irs+1;
    seg.flags = SYN|ACK;
    seg.urg = 0;
    seg.mss = tcpmtu(tcp, lp->laddr, lp->version, &scale);
    seg.wnd = QMAX;

    /* if the other side set scale, we should too */
    if(lp->rcvscale){
        seg.ws = scale;
        lp->sndscale = scale;
    } else {
        seg.ws = 0;
        lp->sndscale = 0;
    }

    switch(lp->version) {
    case V4:
        hbp = htontcp4(&seg, nil, &ph4, nil);
        if(hbp == nil)
            return -1;
        ipoput4(tcp->f, hbp, 0, MAXTTL, DFLTTOS, nil);
        break;
    case V6:
        hbp = htontcp6(&seg, nil, &ph6, nil);
        if(hbp == nil)
            return -1;
        ipoput6(tcp->f, hbp, 0, MAXTTL, DFLTTOS, nil);
        break;
    default:
        panic("sndsnack: version %d", lp->version);
    }
    lp->lastsend = NOW;
    return 0;
}
```

Uses ACK-287 406, DFLTTOS 232b, IP_TCPPROTO-269 406, IP_VER4 94a, IP_VER6 232b, MAXTTL 81g, NOW 234a, QMAX-268 406, SYN-290 406, TCP4_HDRSIZE-272 406, TCP6_HDRSIZE-277 406, V4 21g, V6 21g, htontcp4() 424, htontcp6() 422, ipmove 23c, ipoput4() 93, ipoput6() 486e, tcpmtu() 420a, and v6tov4() 21d.

⟨*macro* hashipa*(*(kernel/network/ip/tcp.c)*)* 431a⟩≡ (464)

```
#define hashipa(a, p) ( ( (a)[IPaddrlen-2] + (a)[IPaddrlen-1] + p )&LHTMASK )
```

⟨*function* limbo 431b⟩≡ (464)

```
/*
 *  put a call into limbo and respond with a SYN ACK
 *
 *  called with proto locked
 */
static void
```

```
limbo(Conv *s, uchar *source, uchar *dest, Tcp *seg, int version)
{
    Limbo *lp, **l;
    Tcppriv *tpriv;
    int h;

    tpriv = s->p->priv;
    h = hashipa(source, seg->source);

    for(l = &tpriv->lht[h]; *l != nil; l = &lp->next){
        lp = *l;
        if(lp->lport != seg->dest || lp->rport != seg->source || lp->version != version)
            continue;
        if(ipcmp(lp->raddr, source) != 0)
            continue;
        if(ipcmp(lp->laddr, dest) != 0)
            continue;

        /* each new SYN restarts the retransmits */
        lp->irs = seg->seq;
        break;
    }
    lp = *l;
    if(lp == nil){
        if(tpriv->nlimbo >= Maxlimbo && tpriv->lht[h]){
            lp = tpriv->lht[h];
            tpriv->lht[h] = lp->next;
            lp->next = nil;
        } else {
            lp = malloc(sizeof(*lp));
            if(lp == nil)
                return;
            tpriv->nlimbo++;
        }
        *l = lp;
        lp->version = version;
        ipmove(lp->laddr, dest);
        ipmove(lp->raddr, source);
        lp->lport = seg->dest;
        lp->rport = seg->source;
        lp->mss = seg->mss;
        lp->rcvscale = seg->ws;
        lp->irs = seg->seq;
        lp->iss = (nrand(1<<16)<<16)|nrand(1<<16);
    }

    if(sndsynack(s->p, lp) < 0){
        *l = lp->next;
        tpriv->nlimbo--;
        free(lp);
    }
}
```

Uses `Maxlimbo-326` 406, `hashipa-362` 431a, `ipcmp` 23d, `ipmove` 23c, and `sndsynack()` 430.

⟨*function* `limborexmit` 432⟩≡ (464)

```
/*
 *  resend SYN ACK's once every SYNACK_RXTIMER ms.
 */
static void
limborexmit(Proto *tcp)
```

```
{
    Tcppriv *tpriv;
    Limbo **l, *lp;
    int h;
    int seen;
    ulong now;

    tpriv = tcp->priv;

    if(!canqlock(tcp))
        return;
    seen = 0;
    now = NOW;
    for(h = 0; h < NLHT && seen < tpriv->nlimbo; h++){
        for(l = &tpriv->lht[h]; *l != nil && seen < tpriv->nlimbo; ){
            lp = *l;
            seen++;
            if(now - lp->lastsend < (lp->rexmits+1)*SYNACK_RXTIMER)
                continue;

            /* time it out after 1 second */
            if(++(lp->rexmits) > 5){
                tpriv->nlimbo--;
                *l = lp->next;
                free(lp);
                continue;
            }

            /* if we're being attacked, don't bother resending SYN ACK's */
            if(tpriv->nlimbo > 100)
                continue;

            if(sndsynack(tcp, lp) < 0){
                tpriv->nlimbo--;
                *l = lp->next;
                free(lp);
                continue;
            }

            l = &lp->next;
        }
    }
    qunlock(tcp);
}
```

Uses NLHT-327 406, NOW 234a, SYNACK_RXTIMER-306 406, and sndsynack() 430.

⟨*function* limborst 433⟩≡ (464)

```
/*
 *  lookup call in limbo.  if found, throw it out.
 *
 *  called with proto locked
 */
static void
limborst(Conv *s, Tcp *segp, uchar *src, uchar *dst, uchar version)
{
    Limbo *lp, **l;
    int h;
    Tcppriv *tpriv;

    tpriv = s->p->priv;
```

```
    /* find a call in limbo */
    h = hashipa(src, segp->source);
    for(l = &tpriv->lht[h]; *l != nil; l = &lp->next){
        lp = *l;
        if(lp->lport != segp->dest || lp->rport != segp->source || lp->version != version)
            continue;
        if(ipcmp(lp->laddr, dst) != 0)
            continue;
        if(ipcmp(lp->raddr, src) != 0)
            continue;

        /* RST can only follow the SYN */
        if(segp->seq == lp->irs+1){
            tpriv->nlimbo--;
            *l = lp->next;
            free(lp);
        }
        break;
    }
}
```

Uses hashipa-362 431a and ipcmp 23d.

⟨*function* initialwindow 434a⟩≡ (464)

```
static void
initialwindow(Tcpctl *tcb)
{
    /* RFC 3390 initial window */
    if(tcb->mss < 1095)
        tcb->cwind = 4*tcb->mss;
    else if(tcb->mss < 2190)
        tcb->cwind = 2*2190;
    else
        tcb->cwind = 2*tcb->mss;
}
```

⟨*function* tcpincoming 434b⟩≡ (464)

```
/*
 *  come here when we finally get an ACK to our SYN-ACK.
 *  lookup call in limbo.  if found, create a new conversation
 *
 *  called with proto locked
 */
static Conv*
tcpincoming(Conv *s, Tcp *segp, uchar *src, uchar *dst, uchar version)
{
    Conv *new;
    Tcpctl *tcb;
    Tcppriv *tpriv;
    Tcp4hdr *h4;
    Tcp6hdr *h6;
    Limbo *lp, **l;
    int h;

    /* unless it's just an ack, it can't be someone coming out of limbo */
    if((segp->flags & SYN) || (segp->flags & ACK) == 0)
        return nil;

    tpriv = s->p->priv;
```

```
/* find a call in limbo */
h = hashipa(src, segp->source);
for(l = &tpriv->lht[h]; (lp = *l) != nil; l = &lp->next){
    netlog(s->p->f, Logtcp, "tcpincoming s %I!%ud/%I!%ud d %I!%ud/%I!%ud v %d/%d\n",
        src, segp->source, lp->raddr, lp->rport,
        dst, segp->dest, lp->laddr, lp->lport,
        version, lp->version
    );

    if(lp->lport != segp->dest || lp->rport != segp->source || lp->version != version)
        continue;
    if(ipcmp(lp->laddr, dst) != 0)
        continue;
    if(ipcmp(lp->raddr, src) != 0)
        continue;

    /* we're assuming no data with the initial SYN */
    if(segp->seq != lp->irs+1 || segp->ack != lp->iss+1){
        netlog(s->p->f, Logtcp, "tcpincoming s %lux/%lux a %lux %lux\n",
            segp->seq, lp->irs+1, segp->ack, lp->iss+1);
        lp = nil;
    } else {
        tpriv->nlimbo--;
        *l = lp->next;
    }
    break;
}
if(lp == nil)
    return nil;

new = Fsnewcall(s, src, segp->source, dst, segp->dest, version);
if(new == nil)
    return nil;

memmove(new->ptcl, s->ptcl, sizeof(Tcpctl));
tcb = (Tcpctl*)new->ptcl;
tcb->flags &= ~CLONE;
tcb->timer.arg = new;
tcb->timer.state = TcptimerOFF;
tcb->acktimer.arg = new;
tcb->acktimer.state = TcptimerOFF;
tcb->katimer.arg = new;
tcb->katimer.state = TcptimerOFF;
tcb->rtt_timer.arg = new;
tcb->rtt_timer.state = TcptimerOFF;

tcb->irs = lp->irs;
tcb->rcv.nxt = tcb->irs+1;
tcb->rcv.wptr = tcb->rcv.nxt;
tcb->rcv.wsnt = 0;
tcb->rcv.urg = tcb->rcv.nxt;

tcb->iss = lp->iss;
tcb->rttseq = tcb->iss;
tcb->snd.wl2 = tcb->iss;
tcb->snd.una = tcb->iss+1;
tcb->snd.ptr = tcb->iss+1;
tcb->snd.nxt = tcb->iss+1;
tcb->snd.rxt = tcb->iss+1;
tcb->flgcnt = 0;
```

```
    tcb->flags |= SYNACK;

    /* our sending max segment size cannot be bigger than what he asked for */
    if(lp->mss != 0 && lp->mss < tcb->mss) {
        tcb->mss = lp->mss;
        tpriv->stats[Mss] = tcb->mss;
    }

    /* window scaling */
    tcpsetscale(new, tcb, lp->rcvscale, lp->sndscale);

    /* congestion window */
    tcb->snd.wnd = segp->wnd;
    initialwindow(tcb);

    /* set initial round trip time */
    tcb->sndsyntime = lp->lastsend+lp->rexmits*SYNACK_RXTIMER;
    tcpsynackrtt(new);

    free(lp);

    /* set up proto header */
    switch(version){
    case V4:
        h4 = &tcb->protohdr.tcp4hdr;
        memset(h4, 0, sizeof(*h4));
        h4->proto = IP_TCPPROTO;
        hnputs(h4->tcpsport, new->lport);
        hnputs(h4->tcpdport, new->rport);
        v6tov4(h4->tcpsrc, dst);
        v6tov4(h4->tcpdst, src);
        break;
    case V6:
        h6 = &tcb->protohdr.tcp6hdr;
        memset(h6, 0, sizeof(*h6));
        h6->proto = IP_TCPPROTO;
        hnputs(h6->tcpsport, new->lport);
        hnputs(h6->tcpdport, new->rport);
        ipmove(h6->tcpsrc, dst);
        ipmove(h6->tcpdst, src);
        break;
    default:
        panic("tcpincoming: version %d", new->ipversion);
    }

    tcpsetstate(new, Established);

    iphtadd(&tpriv->ht, new);

    return new;
}
```

Uses ACK-287 406, CLONE-309 406, Established-319 406, Fsnewcall() 144, IP_TCPPROTO-269 406, Logtcp 233d, Mss-332 410c, SYN-290 406, SYNACK-312 406, SYNACK_RXTIMER-306 406, TcptimerOFF-280 406, V4 21g, V6 21g, hashipa-362 431a, initialwindow() 434a, ipcmp 23d, iphtadd() 136d, ipmove 23c, netlog() 389a, tcpsetscale() 463b, tcpsetstate() 412c, tcpsynackrtt() 437e, and v6tov4() 21d.

⟨*function* seq_within 436⟩≡ (464)

```
static int
seq_within(ulong x, ulong low, ulong high)
{
```

```
    if(low <= high){
        if(low <= x && x <= high)
            return 1;
    }
    else {
        if(x >= low || x <= high)
            return 1;
    }
    return 0;
}
```

⟨*function* `seq_lt` 437a⟩≡ (464)

```
static int
seq_lt(ulong x, ulong y)
{
    return (int)(x-y) < 0;
}
```

⟨*function* `seq_le` 437b⟩≡ (464)

```
static int
seq_le(ulong x, ulong y)
{
    return (int)(x-y) <= 0;
}
```

⟨*function* `seq_gt` 437c⟩≡ (464)

```
static int
seq_gt(ulong x, ulong y)
{
    return (int)(x-y) > 0;
}
```

⟨*function* `seq_ge` 437d⟩≡ (464)

```
static int
seq_ge(ulong x, ulong y)
{
    return (int)(x-y) >= 0;
}
```

⟨*function* `tcpsynackrtt` 437e⟩≡ (464)

```
/*
 *  use the time between the first SYN and it's ack as the
 *  initial round trip time
 */
static void
tcpsynackrtt(Conv *s)
{
    Tcpctl *tcb;
    int delta;
    Tcppriv *tpriv;

    tcb = (Tcpctl*)s->ptcl;
    tpriv = s->p->priv;

    delta = NOW - tcb->sndsyntime;
    tcb->srtt = delta<<LOGAGAIN;
    tcb->mdev = delta<<LOGDGAIN;

    /* halt round trip timer */
    tcphalt(tpriv, &tcb->rtt_timer);
}
```

Uses `LOGAGAIN-313` 406, `LOGDGAIN-314` 406, `NOW` 234a, and `tcphalt()` 419b.

⟨*function* update 438⟩≡ (464)

```
static void
update(Conv *s, Tcp *seg)
{
    int rtt, delta;
    Tcpctl *tcb;
    ulong acked;
    Tcppriv *tpriv;

    if(seg->update)
        return;
    seg->update = 1;

    tpriv = s->p->priv;
    tcb = (Tcpctl*)s->ptcl;

    /* catch zero-window updates, update window & recover */
    if(tcb->snd.wnd == 0 && seg->wnd > 0 &&
        seq_lt(seg->ack, tcb->snd.ptr)){
        netlog(s->p->f, Logtcp, "tcp: zwu ack %lud una %lud ptr %lud win %lud\n",
            seg->ack,  tcb->snd.una, tcb->snd.ptr, seg->wnd);
        tcb->snd.wnd = seg->wnd;
        goto recovery;
    }

    /* newreno fast retransmit */
    if(seg->ack == tcb->snd.una && tcb->snd.una != tcb->snd.nxt &&
        ++tcb->snd.dupacks == 3){       /* was TCPREXMTTHRESH */
recovery:
        if(tcb->snd.recovery){
            tpriv->stats[RecoveryCwind]++;
            tcb->cwind += tcb->mss;
        }else if(seq_le(tcb->snd.rxt, seg->ack)){
            tpriv->stats[Recovery]++;
            tcb->abcbytes = 0;
            tcb->snd.recovery = 1;
            tcb->snd.partialack = 0;
            tcb->snd.rxt = tcb->snd.nxt;
            tcpcongestion(tcb);
            tcb->cwind = tcb->ssthresh + 3*tcb->mss;
            netlog(s->p->f, Logtcpwin, "recovery inflate %ld ss %ld @%lud\n",
                tcb->cwind, tcb->ssthresh, tcb->snd.rxt);
            tcprxmit(s);
        }else{
            tpriv->stats[RecoveryNoSeq]++;
            netlog(s->p->f, Logtcpwin, "!recov %lud not <= %lud %ld\n",
                tcb->snd.rxt, seg->ack, tcb->snd.rxt - seg->ack);
            /* don't enter fast retransmit, don't change ssthresh */
        }
    }else if(tcb->snd.recovery){
        tpriv->stats[RecoveryCwind]++;
        tcb->cwind += tcb->mss;
    }

    /*
     *  update window
     */
    if(seq_gt(seg->ack, tcb->snd.wl2)
    || (tcb->snd.wl2 == seg->ack && seg->wnd > tcb->snd.wnd)){
        /* clear dupack if we advance wl2 */
```

```
    if(tcb->snd.wl2 != seg->ack)
        tcb->snd.dupacks = 0;
    tcb->snd.wnd = seg->wnd;
    tcb->snd.wl2 = seg->ack;
}

if(!seq_gt(seg->ack, tcb->snd.una)){
    /*
     *  don't let us hangup if sending into a closed window and
     *  we're still getting acks
     */
    if((tcb->flags&RETRAN) && tcb->snd.wnd == 0)
        tcb->backedoff = MAXBACKMS/4;
    return;
}

/* Compute the new send window size */
acked = seg->ack - tcb->snd.una;

/* avoid slow start and timers for SYN acks */
if((tcb->flags & SYNACK) == 0) {
    tcb->flags |= SYNACK;
    acked--;
    tcb->flgcnt--;
    goto done;
}

/*
 * congestion control
 */
if(tcb->snd.recovery){
    if(seq_ge(seg->ack, tcb->snd.rxt)){
        /* recovery finished; deflate window */
        tpriv->stats[RecoveryDone]++;
        tcb->snd.dupacks = 0;
        tcb->snd.recovery = 0;
        tcb->cwind = (tcb->snd.nxt - tcb->snd.una) + tcb->mss;
        if(tcb->ssthresh < tcb->cwind)
            tcb->cwind = tcb->ssthresh;
        netlog(s->p->f, Logtcpwin, "recovery deflate %ld %ld\n",
            tcb->cwind, tcb->ssthresh);
    } else {
        /* partial ack; we lost more than one segment */
        tpriv->stats[RecoveryPA]++;
        if(tcb->cwind > acked)
            tcb->cwind -= acked;
        else{
            netlog(s->p->f, Logtcpwin, "partial ack neg\n");
            tcb->cwind = tcb->mss;
        }
        netlog(s->p->f, Logtcpwin, "partial ack %ld left %ld cwind %ld\n",
            acked, tcb->snd.rxt - seg->ack, tcb->cwind);

        if(acked >= tcb->mss)
            tcb->cwind += tcb->mss;
        tcb->snd.partialack++;
    }
} else
    tcpabcincr(tcb, acked);
```

```
    /* Adjust the timers according to the round trip time */
    /* TODO: fix sloppy treatment of overflow cases here. */
    if(tcb->rtt_timer.state == TcptimerON && seq_ge(seg->ack, tcb->rttseq)) {
        tcphalt(tpriv, &tcb->rtt_timer);
        if((tcb->flags&RETRAN) == 0) {
            tcb->backoff = 0;
            tcb->backedoff = 0;
            rtt = tcb->rtt_timer.start - tcb->rtt_timer.count;
            if(rtt == 0)
                rtt = 1; /* else all close sys's will rexmit in 0 time */
            rtt *= MSPTICK;
            if(tcb->srtt == 0) {
                tcb->srtt = rtt << LOGAGAIN;
                tcb->mdev = rtt << LOGDGAIN;
            } else {
                delta = rtt - (tcb->srtt>>LOGAGAIN);
                tcb->srtt += delta;
                if(tcb->srtt <= 0)
                    tcb->srtt = 1;

                delta = abs(delta) - (tcb->mdev>>LOGDGAIN);
                tcb->mdev += delta;
                if(tcb->mdev <= 0)
                    tcb->mdev = 1;
            }
            tcpsettimer(tcb);
        }
    }

done:
    if(qdiscard(s->wq, acked) < acked)
        tcb->flgcnt--;
    tcb->snd.una = seg->ack;

    /* newreno fast recovery */
    if(tcb->snd.recovery)
        tcprxmit(s);

    if(seq_gt(seg->ack, tcb->snd.urg))
        tcb->snd.urg = seg->ack;

    if(tcb->snd.una != tcb->snd.nxt){
        /* `impatient' variant */
        if(!tcb->snd.recovery || tcb->snd.partialack == 1){
            tcb->time = NOW;
            tcb->timeuna = tcb->snd.una;
            tcpgo(tpriv, &tcb->timer);
        }
    } else
        tcphalt(tpriv, &tcb->timer);

    if(seq_lt(tcb->snd.ptr, tcb->snd.una))
        tcb->snd.ptr = tcb->snd.una;

    if(!tcb->snd.recovery)
        tcb->flags &= ~RETRAN;
    tcb->backoff = 0;
    tcb->backedoff = 0;
}
```

Uses LOGAGAIN-313 406, LOGDGAIN-314 406, Logtcp 233d, Logtcpwin 233d, MAXBACKMS-285 406, MSPTICK-299 406, NOW 234a, RETRAN-310 406, Recovery-353 410c, RecoveryCwind-357 410c, RecoveryDone-354 410c, RecoveryNoSeq-356 410c, RecoveryPA-358 410c, SYNACK-312 406, TcptimerON-281 406, netlog() 389a, seq_ge() 437d, seq_gt() 437c, seq_le() 437b, seq_lt() 437a, tcpabcincr() 417b, tcpcongestion() 416b, tcpgo() 419a, tcphalt() 419b, tcprxmit() 455c, and tcpsettimer() 463a.

⟨*function* tcpiput 441⟩≡ (464)

```
static void
tcpiput(Proto *tcp, Ipifc*, Block *bp)
{
    Tcp seg;
    Tcp4hdr *h4;
    Tcp6hdr *h6;
    int hdrlen;
    Tcpctl *tcb;
    ushort length, csum;
    ipaddr source, dest;
    Conv *s;
    Fs *f;
    Tcppriv *tpriv;
    uchar version;

    f = tcp->f;
    tpriv = tcp->priv;

    tpriv->stats[InSegs]++;

    h4 = (Tcp4hdr*)(bp->rp);
    h6 = (Tcp6hdr*)(bp->rp);

    if((h4->vihl&0xF0)==IP_VER4) {
        version = V4;
        length = nhgets(h4->length);
        v4tov6(dest, h4->tcpdst);
        v4tov6(source, h4->tcpsrc);

        h4->Unused = 0;
        hnputs(h4->tcplen, length-TCP4_PKT);
        if(!(bp->flag & Btcpck) && (h4->tcpcksum[0] || h4->tcpcksum[1]) &&
            ptclcsum(bp, TCP4_IPLEN, length-TCP4_IPLEN)) {
            tpriv->stats[CsumErrs]++;
            tpriv->stats[InErrs]++;
            netlog(f, Logtcp, "bad tcp proto cksum\n");
            freeblist(bp);
            return;
        }

        hdrlen = ntohtcp4(&seg, &bp);
        if(hdrlen < 0){
            tpriv->stats[HlenErrs]++;
            tpriv->stats[InErrs]++;
            netlog(f, Logtcp, "bad tcp hdr len\n");
            return;
        }

        /* trim the packet to the size claimed by the datagram */
        length -= hdrlen+TCP4_PKT;
        bp = trimblock(bp, hdrlen+TCP4_PKT, length);
        if(bp == nil){
            tpriv->stats[LenErrs]++;
```

```
            tpriv->stats[InErrs]++;
            netlog(f, Logtcp, "tcp len < 0 after trim\n");
            return;
        }
    }
    else {
        int ttl = h6->ttl;
        int proto = h6->proto;

        version = V6;
        length = nhgets(h6->ploadlen);
        ipmove(dest, h6->tcpdst);
        ipmove(source, h6->tcpsrc);

        h6->ploadlen[0] = h6->ploadlen[1] = h6->proto = 0;
        h6->ttl = proto;
        hnputl(h6->vcf, length);
        if((h6->tcpcksum[0] || h6->tcpcksum[1]) &&
            (csum = ptclcsum(bp, TCP6_IPLEN, length+TCP6_PHDRSIZE)) != 0) {
            tpriv->stats[CsumErrs]++;
            tpriv->stats[InErrs]++;
            netlog(f, Logtcp,
                "bad tcpv6 proto cksum: got %#ux, computed %#ux\n",
                h6->tcpcksum[0]<<8 | h6->tcpcksum[1], csum);
            freeblist(bp);
            return;
        }
        h6->ttl = ttl;
        h6->proto = proto;
        hnputs(h6->ploadlen, length);

        hdrlen = ntohtcp6(&seg, &bp);
        if(hdrlen < 0){
            tpriv->stats[HlenErrs]++;
            tpriv->stats[InErrs]++;
            netlog(f, Logtcp, "bad tcpv6 hdr len\n");
            return;
        }

        /* trim the packet to the size claimed by the datagram */
        length -= hdrlen;
        bp = trimblock(bp, hdrlen+TCP6_PKT, length);
        if(bp == nil){
            tpriv->stats[LenErrs]++;
            tpriv->stats[InErrs]++;
            netlog(f, Logtcp, "tcpv6 len < 0 after trim\n");
            return;
        }
    }

    /* lock protocol while searching for a conversation */
    qlock(tcp);

    /* Look for a matching conversation */
    s = iphtlook(&tpriv->ht, source, seg.source, dest, seg.dest);
    if(s == nil){
        netlog(f, Logtcp, "iphtlook(src %I!%d, dst %I!%d) failed\n",
            source, seg.source, dest, seg.dest);
reset:
        qunlock(tcp);
```

```
    sndrst(tcp, source, dest, length, &seg, version, "no conversation");
    freeblist(bp);
    return;
}

/* if it's a listener, look for the right flags and get a new conv */
tcb = (Tcpctl*)s->ptcl;
if(tcb->state == Listen){
    if(seg.flags & RST){
        limborst(s, &seg, source, dest, version);
        qunlock(tcp);
        freeblist(bp);
        return;
    }

    /* if this is a new SYN, put the call into limbo */
    if((seg.flags & SYN) && (seg.flags & ACK) == 0){
        limbo(s, source, dest, &seg, version);
        qunlock(tcp);
        freeblist(bp);
        return;
    }

    /*
     *  if there's a matching call in limbo, tcpincoming will
     *  return it in state Syn_received
     */
    s = tcpincoming(s, &seg, source, dest, version);
    if(s == nil)
        goto reset;
}

/* The rest of the input state machine is run with the control block
 * locked and implements the state machine directly out of the RFC.
 * Out-of-band data is ignored - it was always a bad idea.
 */
tcb = (Tcpctl*)s->ptcl;
if(waserror()){
    qunlock(s);
    nexterror();
}
qlock(s);
qunlock(tcp);

/* fix up window */
seg.wnd <<= tcb->rcv.scale;

/* every input packet in puts off the keep alive time out */
tcpsetkacounter(tcb);

switch(tcb->state) {
case Closed:
    sndrst(tcp, source, dest, length, &seg, version, "sending to Closed");
    goto raise;
case Syn_sent:
    if(seg.flags & ACK) {
        if(!seq_within(seg.ack, tcb->iss+1, tcb->snd.nxt)) {
            sndrst(tcp, source, dest, length, &seg, version,
                 "bad seq in Syn_sent");
            goto raise;
```

```
            }
        }
        if(seg.flags & RST) {
            if(seg.flags & ACK)
                localclose(s, Econrefused);
            goto raise;
        }

        if(seg.flags & SYN) {
            procsyn(s, &seg);
            if(seg.flags & ACK){
                update(s, &seg);
                tcpsynackrtt(s);
                tcpsetstate(s, Established);
                tcpsetscale(s, tcb, seg.ws, tcb->scale);
            }
            else {
                tcb->time = NOW;
                tcpsetstate(s, Syn_received);   /* DLP - shouldn't this be a reset? */
            }

            if(length != 0 || (seg.flags & FIN))
                break;

            freeblist(bp);
            goto output;
        }
        else
            freeblist(bp);

        qunlock(s);
        poperror();
        return;
    case Syn_received:
        /* doesn't matter if it's the correct ack, we're just trying to set timing */
        if(seg.flags & ACK)
            tcpsynackrtt(s);
        break;
    }

    /*
     *  One DOS attack is to open connections to us and then forget about them,
     *  thereby tying up a conv at no long term cost to the attacker.
     *  This is an attempt to defeat these stateless DOS attacks.  See
     *  corresponding code in tcpsendka().
     */
    if(tcb->state != Syn_received && (seg.flags & RST) == 0){
        if(tcpporthogdefense
        && seq_within(seg.ack, tcb->snd.una-(1<<31), tcb->snd.una-(1<<29))){
            print("stateless hog %I.%d->%I.%d f %ux %lux - %lux - %lux\n",
                source, seg.source, dest, seg.dest, seg.flags,
                tcb->snd.una-(1<<31), seg.ack, tcb->snd.una-(1<<29));
            localclose(s, "stateless hog");
        }
    }

    /* Cut the data to fit the receive window */
    tcprcvwin(s);
    if(tcptrim(tcb, &seg, &bp, &length) == -1) {
        if(seg.seq+1 != tcb->rcv.nxt || length != 1)
```

```
    netlog(f, Logtcp, "tcp: trim: !inwind: seq %lud-%lud win "
        "%lud-%lud l %d from %I\n", seg.seq,
        seg.seq + length - 1, tcb->rcv.nxt,
        tcb->rcv.nxt + tcb->rcv.wnd-1, length, s->raddr);
    update(s, &seg);
    if(qlen(s->wq)+tcb->flgcnt == 0 && tcb->state == Closing) {
        tcphalt(tpriv, &tcb->rtt_timer);
        tcphalt(tpriv, &tcb->acktimer);
        tcphalt(tpriv, &tcb->katimer);
        tcpsetstate(s, Time_wait);
        tcb->timer.start = MSL2*(1000 / MSPTICK);
        tcpgo(tpriv, &tcb->timer);
    }
    if(!(seg.flags & RST)) {
        tcb->flags |= FORCE;
        goto output;
    }
    qunlock(s);
    poperror();
    return;
}

/* Cannot accept so answer with a rst */
if(length && tcb->state == Closed) {
    sndrst(tcp, source, dest, length, &seg, version, "sending to Closed");
    goto raise;
}

/* The segment is beyond the current receive pointer so
 * queue the data in the resequence queue
 */
if(seg.seq != tcb->rcv.nxt)
if(length != 0 || (seg.flags & (SYN|FIN))) {
    update(s, &seg);
    if(addreseq(f, tcb, tpriv, &seg, bp, length) < 0)
        print("reseq %I.%d -> %I.%d\n", s->raddr, s->rport,
            s->laddr, s->lport);
    tcb->flags |= FORCE;    /* force duplicate ack; RFC 5681 §3.2 */
    goto output;
}

if(tcb->nreseq > 0)
    tcb->flags |= FORCE; /* filled hole in seq. space; RFC 5681 §3.2 */

/*
 *  keep looping till we've processed this packet plus any
 *  adjacent packets in the resequence queue
 */
for(;;) {
    if(seg.flags & RST) {
        if(tcb->state == Established) {
            tpriv->stats[EstabResets]++;
            if(tcb->rcv.nxt != seg.seq)
                print("out of order RST rcvd: %I.%d -> "
                    "%I.%d, rcv.nxt %lux seq %lux\n",
                    s->raddr, s->rport, s->laddr,
                    s->lport, tcb->rcv.nxt, seg.seq);
        }
        localclose(s, Econrefused);
        goto raise;
```

```
}

if((seg.flags&ACK) == 0)
    goto raise;

switch(tcb->state) {
case Syn_received:
    if(!seq_within(seg.ack, tcb->snd.una+1, tcb->snd.nxt)){
        sndrst(tcp, source, dest, length, &seg, version,
            "bad seq in Syn_received");
        goto raise;
    }
    update(s, &seg);
    tcpsetstate(s, Established);
case Established:
case Close_wait:
    update(s, &seg);
    break;
case Finwait1:
    update(s, &seg);
    if(qlen(s->wq)+tcb->flgcnt == 0){
        tcphalt(tpriv, &tcb->rtt_timer);
        tcphalt(tpriv, &tcb->acktimer);
        tcpsetkacounter(tcb);
        tcb->time = NOW;
        tcpsetstate(s, Finwait2);
        tcb->katimer.start = MSL2 * (1000 / MSPTICK);
        tcpgo(tpriv, &tcb->katimer);
    }
    break;
case Finwait2:
    update(s, &seg);
    break;
case Closing:
    update(s, &seg);
    if(qlen(s->wq)+tcb->flgcnt == 0) {
        tcphalt(tpriv, &tcb->rtt_timer);
        tcphalt(tpriv, &tcb->acktimer);
        tcphalt(tpriv, &tcb->katimer);
        tcpsetstate(s, Time_wait);
        tcb->timer.start = MSL2*(1000 / MSPTICK);
        tcpgo(tpriv, &tcb->timer);
    }
    break;
case Last_ack:
    update(s, &seg);
    if(qlen(s->wq)+tcb->flgcnt == 0) {
        localclose(s, nil);
        goto raise;
    }
case Time_wait:
    tcb->flags |= FORCE;
    if(tcb->timer.state != TcptimerON)
        tcpgo(tpriv, &tcb->timer);
}

if((seg.flags&URG) && seg.urg) {
    if(seq_gt(seg.urg + seg.seq, tcb->rcv.urg)) {
        tcb->rcv.urg = seg.urg + seg.seq;
        pullblock(&bp, seg.urg);
```

```
	}
}
else
if(seq_gt(tcb->rcv.nxt, tcb->rcv.urg))
	tcb->rcv.urg = tcb->rcv.nxt;

if(length == 0) {
	if(bp != nil)
		freeblist(bp);
}
else {
	switch(tcb->state){
	default:
		/* Ignore segment text */
		if(bp != nil)
			freeblist(bp);
		break;

	case Syn_received:
	case Established:
	case Finwait1:
		/* If we still have some data place on
		 * receive queue
		 */
		if(bp) {
			bp = packblock(bp);
			if(bp == nil)
				panic("tcp packblock");
			qpassnolim(s->rq, bp);
			bp = nil;
		}
		tcb->rcv.nxt += length;

		/*
		 *  turn on the acktimer if there's something
		 *  to ack
		 */
		if(tcb->acktimer.state != TcptimerON)
			tcpgo(tpriv, &tcb->acktimer);

		break;
	case Finwait2:
		/* no process to read the data, send a reset */
		if(bp != nil)
			freeblist(bp);
		sndrst(tcp, source, dest, length, &seg, version,
			"send to Finwait2");
		qunlock(s);
		poperror();
		return;
	}
}

if(seg.flags & FIN) {
	tcb->flags |= FORCE;

	switch(tcb->state) {
	case Syn_received:
	case Established:
		tcb->rcv.nxt++;
```

```
                tcpsetstate(s, Close_wait);
                break;
            case Finwait1:
                tcb->rcv.nxt++;
                if(qlen(s->wq)+tcb->flgcnt == 0) {
                    tcphalt(tpriv, &tcb->rtt_timer);
                    tcphalt(tpriv, &tcb->acktimer);
                    tcphalt(tpriv, &tcb->katimer);
                    tcpsetstate(s, Time_wait);
                    tcb->timer.start = MSL2*(1000/MSPTICK);
                    tcpgo(tpriv, &tcb->timer);
                }
                else
                    tcpsetstate(s, Closing);
                break;
            case Finwait2:
                tcb->rcv.nxt++;
                tcphalt(tpriv, &tcb->rtt_timer);
                tcphalt(tpriv, &tcb->acktimer);
                tcphalt(tpriv, &tcb->katimer);
                tcpsetstate(s, Time_wait);
                tcb->timer.start = MSL2 * (1000/MSPTICK);
                tcpgo(tpriv, &tcb->timer);
                break;
            case Close_wait:
            case Closing:
            case Last_ack:
                break;
            case Time_wait:
                tcpgo(tpriv, &tcb->timer);
                break;
            }
        }

        /*
         *  get next adjacent segment from the resequence queue.
         *  dump/trim any overlapping segments
         */
        for(;;) {
            if(tcb->reseq == nil)
                goto output;

            if(seq_ge(tcb->rcv.nxt, tcb->reseq->seg.seq) == 0)
                goto output;

            getreseq(tcb, &seg, &bp, &length);

            tcprcvwin(s);
            if(tcptrim(tcb, &seg, &bp, &length) == 0){
                tcb->flags |= FORCE;
                break;
            }
        }
    }
output:
    tcpoutput(s);
    qunlock(s);
    poperror();
    return;
raise:
```

```
        qunlock(s);
        poperror();
        freeblist(bp);
        tcpkick(s);
    }
```

Uses ACK-287 406, Close_wait-322 406, Closed-315 406, Closing-323 406, CsumErrs-344 410c, EstabResets-335 410c, Established-319 406, FIN-291 406, FORCE-308 406, Finwait1-320 406, Finwait2-321 406, HlenErrs-345 410c, IP_VER4 94a, InErrs-342 410c, InSegs-337 410c, Last_ack-324 406, LenErrs-346 410c, Listen-316 406, Logtcp 233d, MSL2-298 406, MSPTICK-299 406, NOW 234a, RST-289 406, SYN-290 406, Syn_received-318 406, Syn_sent-317 406, TCP4_IPLEN-270 406, TCP4_PKT-274 406, TCP6_IPLEN-275 406, TCP6_PHDRSIZE-276 406, TCP6_PKT-279 406, TcptimerON-281 406, Time_wait-325 406, URG-286 406, V4 21g, V6 21g, addreseq() 458c, getreseq() 459a, iphtlook() 143, ipmove 23c, limbo() 431b, limborst() 433, localclose() 419d, netlog() 389a, ntohtcp4() 426, ntohtcp6() 425, procsyn() 457b, ptclcsum() 90d, seq_ge() 437d, seq_gt() 437c, seq_within() 436, sndrst() 428, tcpgo() 419a, tcphalt() 419b, tcpincoming() 434b, tcpkick() 415a, tcpoutput() 449, tcpporthogdefense 412b, tcprcvwin() 415b, tcpsetkacounter() 454a, tcpsetscale() 463b, tcpsetstate() 412c, tcpsynackrtt() 437e, tcptrim() 459b, update() 438, and v4tov6() 21c.

⟨*function* tcpoutput 449⟩≡ (464)

```
  /*
   *  always enters and exits with the s locked.  We drop
   *  the lock to ipoput the packet so some care has to be
   *  taken by callers.
   */
  static void
  tcpoutput(Conv *s)
  {
      Tcp seg;
      uint msgs;
      Tcpctl *tcb;
      Block *hbp, *bp;
      int sndcnt;
      ulong ssize, dsize, sent;
      Fs *f;
      Tcppriv *tpriv;
      uchar version;

      f = s->p->f;
      tpriv = s->p->priv;
      version = s->ipversion;

      tcb = (Tcpctl*)s->ptcl;

      /* force ack every 2*mss */
      if((tcb->flags & FORCE) == 0 &&
          tcb->rcv.nxt - tcb->rcv.ackptr >= 2*tcb->mss){
          tpriv->stats[Delayack]++;
          tcb->flags |= FORCE;
      }

      /* force ack if window opening */
      if((tcb->flags & FORCE) == 0){
          tcprcvwin(s);
          if((int)(tcb->rcv.wptr - tcb->rcv.wsnt) >= 2*tcb->mss){
              tpriv->stats[Wopenack]++;
              tcb->flags |= FORCE;
          }
      }

      for(msgs = 0; msgs < 100; msgs++) {
          switch(tcb->state) {
          case Listen:
```

```
case Closed:
case Finwait2:
    return;
}

/* Don't send anything else until our SYN has been acked */
if(tcb->snd.ptr != tcb->iss && (tcb->flags & SYNACK) == 0)
    break;

/* force an ack when a window has opened up */
tcprcvwin(s);
if(tcb->rcv.blocked && tcb->rcv.wnd > 0){
    tcb->rcv.blocked = 0;
    tcb->flags |= FORCE;
}

sndcnt = qlen(s->wq)+tcb->flgcnt;
sent = tcb->snd.ptr - tcb->snd.una;
ssize = sndcnt;
if(tcb->snd.wnd == 0){
    /* zero window probe */
    if(sent > 0 && !(tcb->flags & FORCE))
        break;  /* already probing, rto re-probes */
    if(ssize < sent)
        ssize = 0;
    else{
        ssize -= sent;
        if(ssize > 0)
            ssize = 1;
    }
} else {
    /* calculate usable segment size */
    if(ssize > tcb->cwind)
        ssize = tcb->cwind;
    if(ssize > tcb->snd.wnd)
        ssize = tcb->snd.wnd;

    if(ssize < sent)
        ssize = 0;
    else {
        ssize -= sent;
        if(ssize > tcb->mss)
            ssize = tcb->mss;
    }
}

dsize = ssize;
seg.urg = 0;

if(!(tcb->flags & FORCE))
    if(ssize == 0 ||
        ssize < tcb->mss && tcb->snd.nxt == tcb->snd.ptr &&
        sent > TCPREXMTTHRESH * tcb->mss)
        break;

tcb->flags &= ~FORCE;

/* By default we will generate an ack */
tcphalt(tpriv, &tcb->acktimer);
seg.source = s->lport;
```

```
seg.dest = s->rport;
seg.flags = ACK;
seg.mss = 0;
seg.ws = 0;
seg.update = 0;
switch(tcb->state){
case Syn_sent:
    seg.flags = 0;
    if(tcb->snd.ptr == tcb->iss){
        seg.flags |= SYN;
        dsize--;
        seg.mss = tcb->mss;
        seg.ws = tcb->scale;
    }
    break;
case Syn_received:
    /*
     *  don't send any data with a SYN/ACK packet
     *  because Linux rejects the packet in its
     *  attempt to solve the SYN attack problem
     */
    if(tcb->snd.ptr == tcb->iss){
        seg.flags |= SYN;
        dsize = 0;
        ssize = 1;
        seg.mss = tcb->mss;
        seg.ws = tcb->scale;
    }
    break;
}
seg.seq = tcb->snd.ptr;
seg.ack = tcb->rcv.nxt;
seg.wnd = tcb->rcv.wnd;

/* Pull out data to send */
bp = nil;
if(dsize != 0) {
    bp = qcopy(s->wq, dsize, sent);
    if(BLEN(bp) != dsize) {
        seg.flags |= FIN;
        dsize--;
    }
}

if(sent+dsize == sndcnt && dsize)
    seg.flags |= PSH;

tcb->snd.ptr += ssize;

/* Pull up the send pointer so we can accept acks
 * for this window
 */
if(seq_gt(tcb->snd.ptr,tcb->snd.nxt))
    tcb->snd.nxt = tcb->snd.ptr;

/* Build header, link data and compute cksum */
switch(version){
case V4:
    tcb->protohdr.tcp4hdr.vihl = IP_VER4;
    hbp = htontcp4(&seg, bp, &tcb->protohdr.tcp4hdr, tcb);
```

```
    if(hbp == nil) {
        freeblist(bp);
        return;
    }
    break;
case V6:
    tcb->protohdr.tcp6hdr.vcf[0] = IP_VER6;
    hbp = htontcp6(&seg, bp, &tcb->protohdr.tcp6hdr, tcb);
    if(hbp == nil) {
        freeblist(bp);
        return;
    }
    break;
default:
    hbp = nil;  /* to suppress a warning */
    panic("tcpoutput: version %d", version);
}

/* Start the transmission timers if there is new data and we
 * expect acknowledges
 */
if(ssize != 0){
    if(tcb->timer.state != TcptimerON){
        tcb->time = NOW;
        tcb->timeuna = tcb->snd.una;
        tcpgo(tpriv, &tcb->timer);
    }

    /*  If round trip timer isn't running, start it.
     *  measure the longest packet only in case the
     *  transmission time dominates RTT
     */
    if(tcb->snd.retransmit == 0)
    if(tcb->rtt_timer.state != TcptimerON)
    if(ssize == tcb->mss) {
        tcpgo(tpriv, &tcb->rtt_timer);
        tcb->rttseq = tcb->snd.ptr;
    }
}

tpriv->stats[OutSegs]++;
if(tcb->snd.retransmit)
    tpriv->stats[RetransSegsSent]++;
tcb->rcv.ackptr = seg.ack;
tcb->rcv.wsnt = tcb->rcv.wptr;

/* put off the next keep alive */
tcpgo(tpriv, &tcb->katimer);

switch(version){
case V4:
    if(ipoput4(f, hbp, 0, s->ttl, s->tos, s) < 0){
        /* a negative return means no route */
        localclose(s, "no route");
    }
    break;
case V6:
    if(ipoput6(f, hbp, 0, s->ttl, s->tos, s) < 0){
        /* a negative return means no route */
        localclose(s, "no route");
```

```
            }
            break;
        default:
            panic("tcpoutput2: version %d", version);
        }
        if((msgs%4) == 3){
            qunlock(s);
            qlock(s);
        }
    }
}
```

Uses ACK-287 406, Closed-315 406, Delayack-351 410c, FIN-291 406, FORCE-308 406, Finwait2-321 406, IP_VER4 94a, IP_VER6 232b, Listen-316 406, NOW 234a, OutSegs-338 410c, PSH-288 406, RetransSegsSent-340 410c, SYN-290 406, SYNACK-312 406, Syn_received-318 406, Syn_sent-317 406, TCPREXMTTHRESH-307 406, TcptimerON-281 406, V4 21g, V6 21g, Wopenack-352 410c, htontcp4() 424, htontcp6() 422, ipoput4() 93, ipoput6() 486e, localclose() 419d, seq_gt() 437c, tcpgo() 419a, tcphalt() 419b, and tcprcvwin() 415b.

⟨*function* tcpsendka 453⟩≡ (464)

```
/*
 *  the BSD convention (hack?) for keep alives.  resend last uchar acked.
 */
static void
tcpsendka(Conv *s)
{
    Tcp seg;
    Tcpctl *tcb;
    Block *hbp,*dbp;

    tcb = (Tcpctl*)s->ptcl;

    dbp = nil;
    memset(&seg, 0, sizeof seg);
    seg.urg = 0;
    seg.source = s->lport;
    seg.dest = s->rport;
    seg.flags = ACK|PSH;
    seg.mss = 0;
    seg.ws = 0;
    if(tcpporthogdefense)
        seg.seq = tcb->snd.una-(1<<30)-nrand(1<<20);
    else
        seg.seq = tcb->snd.una-1;
    seg.ack = tcb->rcv.nxt;
    tcb->rcv.ackptr = seg.ack;
    tcprcvwin(s);
    seg.wnd = tcb->rcv.wnd;
    if(tcb->state == Finwait2){
        seg.flags |= FIN;
    } else {
        dbp = allocb(1);
        dbp->wp++;
    }

    if(isv4(s->raddr)) {
        /* Build header, link data and compute cksum */
        tcb->protohdr.tcp4hdr.vihl = IP_VER4;
        hbp = htontcp4(&seg, dbp, &tcb->protohdr.tcp4hdr, tcb);
        if(hbp == nil) {
            freeblist(dbp);
            return;
```

```
        }
        ipoput4(s->p->f, hbp, 0, s->ttl, s->tos, s);
    }
    else {
        /* Build header, link data and compute cksum */
        tcb->protohdr.tcp6hdr.vcf[0] = IP_VER6;
        hbp = htontcp6(&seg, dbp, &tcb->protohdr.tcp6hdr, tcb);
        if(hbp == nil) {
            freeblist(dbp);
            return;
        }
        ipoput6(s->p->f, hbp, 0, s->ttl, s->tos, s);
    }
}
```

Uses ACK-287 406, FIN-291 406, Finwait2-321 406, IP_VER4 94a, IP_VER6 232b, PSH-288 406, htontcp4() 424, htontcp6() 422, ipoput4() 93, ipoput6() 486e, isv4() 21b, tcpporthogdefense 412b, and tcprcvwin() 415b.

⟨*function* tcpsetkacounter 454a⟩≡ (464)

```
/*
 *  set connection to time out after 12 minutes
 */
static void
tcpsetkacounter(Tcpctl *tcb)
{
    tcb->kacounter = (12 * 60 * 1000) / (tcb->katimer.start*MSPTICK);
    if(tcb->kacounter < 3)
        tcb->kacounter = 3;
}
```

Uses MSPTICK-299 406.

⟨*function* tcpkeepalive 454b⟩≡ (464)

```
/*
 *  if we've timed out, close the connection
 *  otherwise, send a keepalive and restart the timer
 */
static void
tcpkeepalive(void *v)
{
    Tcpctl *tcb;
    Conv *s;

    s = v;
    tcb = (Tcpctl*)s->ptcl;
    if(waserror()){
        qunlock(s);
        nexterror();
    }
    qlock(s);
    if(tcb->state != Closed){
        if(--(tcb->kacounter) <= 0) {
            localclose(s, Etimedout);
        } else {
            tcpsendka(s);
            tcpgo(s->p->priv, &tcb->katimer);
        }
    }
    qunlock(s);
    poperror();
}
```

Uses Closed-315 406, localclose() 419d, tcpgo() 419a, and tcpsendka() 453.

⟨*function* tcpstartka 455a⟩≡ (464)

```
/*
 *  start keepalive timer
 */
static char*
tcpstartka(Conv *s, char **f, int n)
{
    Tcpctl *tcb;
    int x;

    tcb = (Tcpctl*)s->ptcl;
    if(tcb->state != Established)
        return "connection must be in Establised state";
    if(n > 1){
        x = atoi(f[1]);
        if(x >= MSPTICK)
            tcb->katimer.start = x/MSPTICK;
    }
    tcpsetkacounter(tcb);
    tcpgo(s->p->priv, &tcb->katimer);

    return nil;
}
```

Uses Established-319 406, MSPTICK-299 406, tcpgo() 419a, and tcpsetkacounter() 454a.

⟨*function* tcpsetchecksum 455b⟩≡ (464)

```
/*
 *  turn checksums on/off
 */
static char*
tcpsetchecksum(Conv *s, char **f, int)
{
    Tcpctl *tcb;

    tcb = (Tcpctl*)s->ptcl;
    tcb->nochecksum = !atoi(f[1]);

    return nil;
}
```

⟨*function* tcprxmit 455c⟩≡ (464)

```
/*
 *  retransmit (at most) one segment at snd.una.
 *  preserve cwind & snd.ptr
 */
static void
tcprxmit(Conv *s)
{
    Tcpctl *tcb;
    Tcppriv *tpriv;
    ulong tcwind, tptr;

    tcb = (Tcpctl*)s->ptcl;
    tcb->flags |= RETRAN|FORCE;

    tptr = tcb->snd.ptr;
    tcwind = tcb->cwind;
    tcb->snd.ptr = tcb->snd.una;
    tcb->cwind = tcb->mss;
    tcb->snd.retransmit = 1;
```

```
    tcpoutput(s);
    tcb->snd.retransmit = 0;
    tcb->cwind = tcwind;
    tcb->snd.ptr = tptr;

    tpriv = s->p->priv;
    tpriv->stats[RetransSegs]++;
}
```

Uses FORCE-308 406, RETRAN-310 406, RetransSegs-339 410c, and tcpoutput() 449.

⟨*function* tcptimeout 456⟩≡ (464)

```
/*
 *  TODO: RFC 4138 F-RTO
 */
static void
tcptimeout(void *arg)
{
    Conv *s;
    Tcpctl *tcb;
    int maxback;
    Tcppriv *tpriv;

    s = (Conv*)arg;
    tpriv = s->p->priv;
    tcb = (Tcpctl*)s->ptcl;

    if(waserror()){
        qunlock(s);
        nexterror();
    }
    qlock(s);
    switch(tcb->state){
    default:
        tcb->backoff++;
        if(tcb->state == Syn_sent)
            maxback = MAXBACKMS/2;
        else
            maxback = MAXBACKMS;
        tcb->backedoff += tcb->timer.start * MSPTICK;
        if(tcb->backedoff >= maxback) {
            localclose(s, Etimedout);
            break;
        }
        netlog(s->p->f, Logtcprxmt, "rxm %d/%d %ldms %lud rto %d %lud %s\n",
            tcb->srtt, tcb->mdev, NOW - tcb->time,
            tcb->snd.una - tcb->timeuna, tcb->snd.rto, tcb->snd.ptr,
            tcpstates[s->state]);
        tcpsettimer(tcb);
        if(tcb->snd.rto == 0)
            tcpcongestion(tcb);
        tcprxmit(s);
        tcb->snd.ptr = tcb->snd.una;
        tcb->cwind = tcb->mss;
        tcb->snd.rto = 1;
        tpriv->stats[RetransTimeouts]++;

        if(tcb->snd.recovery){
            tcb->snd.dupacks = 0;       /* reno rto */
            tcb->snd.recovery = 0;
            tpriv->stats[RecoveryRTO]++;
```

```
            tcb->snd.rxt = tcb->snd.nxt;
            netlog(s->p->f, Logtcpwin,
                "rto recovery rxt @%lud\n", tcb->snd.nxt);
        }

        tcb->abcbytes = 0;
        break;
    case Time_wait:
        localclose(s, nil);
        break;
    case Closed:
        break;
    }
    qunlock(s);
    poperror();
}
```

Uses Closed-315 406, Logtcprxmt 233d, Logtcpwin 233d, MAXBACKMS-285 406, MSPTICK-299 406, NOW 234a, RecoveryRTO-355 410c, RetransTimeouts-341 410c, Syn_sent-317 406, Time_wait-325 406, localclose() 419d, netlog() 389a, tcpcongestion() 416b, tcprxmit() 455c, tcpsettimer() 463a, and tcpstates 407a.

⟨*function* inwindow 457a⟩≡ (464)

```
static int
inwindow(Tcpctl *tcb, int seq)
{
    return seq_within(seq, tcb->rcv.nxt, tcb->rcv.nxt+tcb->rcv.wnd-1);
}
```

Uses seq_within() 436.

⟨*function* procsyn 457b⟩≡ (464)

```
/*
 *  set up state for a received SYN (or SYN ACK) packet
 */
static void
procsyn(Conv *s, Tcp *seg)
{
    Tcpctl *tcb;
    Tcppriv *tpriv;

    tcb = (Tcpctl*)s->ptcl;
    tcb->flags |= FORCE;

    tcb->rcv.nxt = seg->seq + 1;
    tcb->rcv.wptr = tcb->rcv.nxt;
    tcb->rcv.wsnt = 0;
    tcb->rcv.urg = tcb->rcv.nxt;
    tcb->irs = seg->seq;

    /* our sending max segment size cannot be bigger than what he asked for */
    if(seg->mss != 0 && seg->mss < tcb->mss) {
        tcb->mss = seg->mss;
        tpriv = s->p->priv;
        tpriv->stats[Mss] = tcb->mss;
    }

    tcb->snd.wnd = seg->wnd;
    initialwindow(tcb);
}
```

Uses FORCE-308 406, Mss-332 410c, and initialwindow() 434a.

⟨*function* dumpreseq 458a⟩≡ (464)

```
static int
dumpreseq(Tcpctl *tcb)
{
    Reseq *r, *next;

    for(r = tcb->reseq; r != nil; r = next){
        next = r->next;
        freeblist(r->bp);
        free(r);
    }
    tcb->reseq = nil;
    tcb->nreseq = 0;
    tcb->reseqlen = 0;
    return -1;
}
```

⟨*function* logreseq 458b⟩≡ (464)

```
static void
logreseq(Fs *f, Reseq *r, ulong n)
{
    char *s;

    for(; r != nil; r = r->next){
        s = nil;
        if(r->next == nil && r->seg.seq != n)
            s = "hole/end";
        else if(r->next == nil)
            s = "end";
        else if(r->seg.seq != n)
            s = "hole";
        if(s != nil)
            netlog(f, Logtcp, "%s %lud-%lud (%ld) %#ux\n", s,
                n, r->seg.seq, r->seg.seq - n, r->seg.flags);
        n = r->seg.seq + r->seg.len;
    }
}
```

Uses `Logtcp` 233d and `netlog()` 389a.

⟨*function* addreseq 458c⟩≡ (464)

```
static int
addreseq(Fs *f, Tcpctl *tcb, Tcppriv *tpriv, Tcp *seg, Block *bp, ushort length)
{
    Reseq *rp, **rr;
    int qmax;

    rp = malloc(sizeof *rp);
    if(rp == nil){
        freeblist(bp);      /* bp always consumed by addreseq */
        return 0;
    }

    rp->seg = *seg;
    rp->bp = bp;
    rp->length = length;

    tcb->reseqlen += length;
    tcb->nreseq++;

    /* Place on reassembly list sorting by starting seq number */
```

```
    for(rr = &tcb->reseq; ; rr = &(*rr)->next)
        if(*rr == nil || seq_lt(seg->seq, (*rr)->seg.seq)){
            rp->next = *rr;
            *rr = rp;
            tpriv->stats[Resequenced]++;
            if(rp->next != nil)
                tpriv->stats[OutOfOrder]++;
            break;
        }

    qmax = tcb->window;
    if(tcb->reseqlen > qmax){
        netlog(f, Logtcp, "tcp: reseq: queue > window: %d > %d; %d packets\n",
            tcb->reseqlen, qmax, tcb->nreseq);
        logreseq(f, tcb->reseq, tcb->rcv.nxt);
        tpriv->stats[ReseqBytelim]++;
        return dumpreseq(tcb);
    }
    qmax = tcb->window / tcb->mss; /* ~190 for qscale=2, 390 for qscale=3 */
    if(tcb->nreseq > qmax){
        netlog(f, Logtcp, "resequence queue > packets: %d %d; %d bytes\n",
            tcb->nreseq, qmax, tcb->reseqlen);
        logreseq(f, tcb->reseq, tcb->rcv.nxt);
        tpriv->stats[ReseqPktlim]++;
        return dumpreseq(tcb);
    }
    return 0;
}
```

Uses Logtcp 233d, OutOfOrder-348 410c, ReseqBytelim-349 410c, ReseqPktlim-350 410c, Resequenced-347 410c, dumpreseq() 458a, logreseq() 458b, netlog() 389a, and seq_lt() 437a.

⟨*function* getreseq 459a⟩≡ (464)

```
static void
getreseq(Tcpctl *tcb, Tcp *seg, Block **bp, ushort *length)
{
    Reseq *rp;

    rp = tcb->reseq;
    if(rp == nil)
        return;

    tcb->reseq = rp->next;

    *seg = rp->seg;
    *bp = rp->bp;
    *length = rp->length;

    tcb->nreseq--;
    tcb->reseqlen -= rp->length;

    free(rp);
}
```

⟨*function* tcptrim 459b⟩≡ (464)

```
static int
tcptrim(Tcpctl *tcb, Tcp *seg, Block **bp, ushort *length)
{
    ushort len;
    uchar accept;
    int dupcnt, excess;
```

```
accept = 0;
len = *length;
if(seg->flags & SYN)
    len++;
if(seg->flags & FIN)
    len++;

if(tcb->rcv.wnd == 0) {
    if(len == 0 && seg->seq == tcb->rcv.nxt)
        return 0;
}
else {
    /* Some part of the segment should be in the window */
    if(inwindow(tcb,seg->seq))
        accept++;
    else
    if(len != 0) {
        if(inwindow(tcb, seg->seq+len-1) ||
        seq_within(tcb->rcv.nxt, seg->seq,seg->seq+len-1))
            accept++;
    }
}
if(!accept) {
    freeblist(*bp);
    return -1;
}
dupcnt = tcb->rcv.nxt - seg->seq;
if(dupcnt > 0){
    tcb->rerecv += dupcnt;
    if(seg->flags & SYN){
        seg->flags &= ~SYN;
        seg->seq++;

        if(seg->urg > 1)
            seg->urg--;
        else
            seg->flags &= ~URG;
        dupcnt--;
    }
    if(dupcnt > 0){
        pullblock(bp, (ushort)dupcnt);
        seg->seq += dupcnt;
        *length -= dupcnt;

        if(seg->urg > dupcnt)
            seg->urg -= dupcnt;
        else {
            seg->flags &= ~URG;
            seg->urg = 0;
        }
    }
}
excess = seg->seq + *length - (tcb->rcv.nxt + tcb->rcv.wnd);
if(excess > 0) {
    tcb->rerecv += excess;
    *length -= excess;
    *bp = trimblock(*bp, 0, *length);
    if(*bp == nil)
        panic("presotto is a boofhead");
```

```
        seg->flags &= ~FIN;
    }
    return 0;
}
```

Uses FIN-291 406, SYN-290 406, URG-286 406, inwindow() 457a, and seq_within() 436.

⟨*function* tcpadvise 461⟩≡ (464)

```
static void
tcpadvise(Proto *tcp, Block *bp, char *msg)
{
    Tcp4hdr *h4;
    Tcp6hdr *h6;
    Tcpctl *tcb;
    ipaddr source, dest;
    ushort psource, pdest;
    Conv *s, **p;

    h4 = (Tcp4hdr*)(bp->rp);
    h6 = (Tcp6hdr*)(bp->rp);

    if((h4->vihl&0xF0)==IP_VER4) {
        v4tov6(dest, h4->tcpdst);
        v4tov6(source, h4->tcpsrc);
        psource = nhgets(h4->tcpsport);
        pdest = nhgets(h4->tcpdport);
    }
    else {
        ipmove(dest, h6->tcpdst);
        ipmove(source, h6->tcpsrc);
        psource = nhgets(h6->tcpsport);
        pdest = nhgets(h6->tcpdport);
    }

    /* Look for a connection */
    qlock(tcp);
    for(p = tcp->conv; *p; p++) {
        s = *p;
        tcb = (Tcpctl*)s->ptcl;
        if(s->rport == pdest)
        if(s->lport == psource)
        if(tcb->state != Closed)
        if(ipcmp(s->raddr, dest) == 0)
        if(ipcmp(s->laddr, source) == 0){
            qlock(s);
            qunlock(tcp);
            switch(tcb->state){
            case Syn_sent:
                localclose(s, msg);
                break;
            }
            qunlock(s);
            freeblist(bp);
            return;
        }
    }
    qunlock(tcp);
    freeblist(bp);
}
```

Uses Closed-315 406, IP_VER4 94a, Syn_sent-317 406, ipcmp 23d, ipmove 23c, localclose() 419d, and v4tov6() 21c.

⟨*function* tcpporthogdefensectl 462a⟩≡ (464)
```
static char*
tcpporthogdefensectl(char *val)
{
    if(strcmp(val, "on") == 0)
        tcpporthogdefense = 1;
    else if(strcmp(val, "off") == 0)
        tcpporthogdefense = 0;
    else
        return "unknown value for tcpporthogdefense";
    return nil;
}
```
Uses tcpporthogdefense 412b.

⟨*function* tcpctl 462b⟩≡ (464)
```
/* called with c qlocked */
static char*
tcpctl(Conv* c, char** f, int n)
{
    if(n == 1 && strcmp(f[0], "hangup") == 0)
        return tcphangup(c);
    if(n >= 1 && strcmp(f[0], "keepalive") == 0)
        return tcpstartka(c, f, n);
    if(n >= 1 && strcmp(f[0], "checksum") == 0)
        return tcpsetchecksum(c, f, n);
    if(n >= 1 && strcmp(f[0], "tcpporthogdefense") == 0)
        return tcpporthogdefensectl(f[1]);
    return "unknown control request";
}
```
Uses tcphangup() 429, tcpporthogdefensectl() 462a, tcpsetchecksum() 455b, and tcpstartka() 455a.

⟨*function* tcpstats 462c⟩≡ (464)
```
static int
tcpstats(Proto *tcp, char *buf, int len)
{
    Tcppriv *priv;
    char *p, *e;
    int i;

    priv = tcp->priv;
    p = buf;
    e = p+len;
    for(i = 0; i < Nstats; i++)
        p = seprint(p, e, "%s: %llud\n", statnames[i], priv->stats[i]);
    return p - buf;
}
```
Uses Nstats-359 410c and statnames-360 411.

⟨*function* tcpgc 462d⟩≡ (464)
```
/*
 *  garbage collect any stale conversations:
 *  - SYN received but no SYN-ACK after 5 seconds (could be the SYN attack)
 *  - Finwait2 after 5 minutes
 *
 *  this is called whenever we run out of channels.  Both checks are
 *  of questionable validity so we try to use them only when we're
 *  up against the wall.
 */
static int
```

```
tcpgc(Proto *tcp)
{
    Conv *c, **pp, **ep;
    int n;
    Tcpctl *tcb;


    n = 0;
    ep = &tcp->conv[tcp->nc];
    for(pp = tcp->conv; pp < ep; pp++) {
        c = *pp;
        if(c == nil)
            break;
        if(!canqlock(c))
            continue;
        tcb = (Tcpctl*)c->ptcl;
        switch(tcb->state){
        case Syn_received:
            if(NOW - tcb->time > 5000){
                localclose(c, Etimedout);
                n++;
            }
            break;
        case Finwait2:
            if(NOW - tcb->time > 5*60*1000){
                localclose(c, Etimedout);
                n++;
            }
            break;
        }
        qunlock(c);
    }
    return n;
}
```

Uses `Finwait2-321` 406, `NOW` 234a, `Syn_received-318` 406, and `localclose()` 419d.

⟨*function* `tcpsettimer` 463a⟩≡ (464)

```
static void
tcpsettimer(Tcpctl *tcb)
{
    int x;

    /* round trip dependency */
    x = backoff(tcb->backoff) *
        (tcb->mdev + (tcb->srtt>>LOGAGAIN) + MSPTICK) / MSPTICK;

    /* bounded twixt 0.3 and 64 seconds */
    if(x < 300/MSPTICK)
        x = 300/MSPTICK;
    else if(x > (64000/MSPTICK))
        x = 64000/MSPTICK;
    tcb->timer.start = x;
}
```

Uses `LOGAGAIN-313` 406, `MSPTICK-299` 406, and `backoff()` 419c.

⟨*function* `tcpsetscale` 463b⟩≡ (464)

```
static void
tcpsetscale(Conv *s, Tcpctl *tcb, ushort rcvscale, ushort sndscale)
{
    /*
```

```
     * guess at reasonable queue sizes.  there's no current way
     * to know how many nic receive buffers we can safely tie up in the
     * tcp stack, and we don't adjust our queues to maximize throughput
     * and minimize bufferbloat.  n.b. the offer (rcvscale) needs to be
     * respected, but we still control our own buffer commitment by
     * keeping a seperate qscale.
     */
    tcb->rcv.scale = rcvscale & 0xff;
    tcb->snd.scale = sndscale & 0xff;
    tcb->qscale = rcvscale & 0xff;
    if(rcvscale > Maxqscale)
        tcb->qscale = Maxqscale;

    if(rcvscale != tcb->rcv.scale)
        netlog(s->p->f, Logtcp, "tcpsetscale: window %lud "
            "qlen %d >> window %ud lport %d\n",
            tcb->window, qlen(s->rq), QMAX<<tcb->qscale, s->lport);
    tcb->window = QMAX << tcb->qscale;
    tcb->ssthresh = tcb->window;

    /*
     * it's important to set wq large enough to cover the full
     * bandwidth-delay product.  it's possible to be in loss
     * recovery with a big window, and we need to keep sending
     * into the inflated window.  the difference can be huge
     * for even modest (70ms) ping times.
     */
    qsetlimit(s->rq, tcb->window);
    qsetlimit(s->wq, tcb->window);
    tcprcvwin(s);
}
```

Uses `Logtcp` 233d, `Maxqscale-329` 406, `QMAX-268` 406, `netlog()` 389a, and `tcprcvwin()` 415b.

⟨`kernel/network/ip/tcp.c` 464⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "../port/error.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"

#include    "ip.h"

typedef struct Tcptimer Tcptimer;
typedef struct Tcp4hdr Tcp4hdr;
typedef struct Tcp6hdr Tcp6hdr;
typedef struct Tcp Tcp;
typedef struct Reseq Reseq;
typedef struct Tcpctl Tcpctl;
typedef struct Limbo Limbo;
typedef struct Tcppriv Tcppriv;

⟨enum _anon_ (kernel/network/ip/tcp.c) 406⟩

⟨global tcpstates 407a⟩

⟨struct Tcptimer 407b⟩

/*
 *  v4 and v6 pseudo headers used for
```

```
 *  checksuming tcp
 */
⟨struct Tcp4hdr 407c⟩

⟨struct Tcp6hdr 408a⟩

/*
 *  this represents the control info
 *  for a single packet.  It is derived from
 *  a packet in ntohtcp{4,6}() and stuck into
 *  a packet in htontcp{4,6}().
 */
⟨struct Tcp 408b⟩

/*
 *  this header is malloc'd to thread together fragments
 *  waiting to be coalesced
 */
⟨struct Reseq 409a⟩

/*
 *  the qlock in the Conv locks this structure
 */
⟨struct Tcpctl 409b⟩

/*
 *  New calls are put in limbo rather than having a conversation structure
 *  allocated.  Thus, a SYN attack results in lots of limbo'd calls but not
 *  any real Conv structures mucking things up.  Calls in limbo rexmit their
 *  SYN ACK every SYNACK_RXTIMER ms up to 4 times, i.e., they disappear after 1 second.
 *
 *  In particular they aren't on a listener's queue so that they don't figure
 *  in the input queue limit.
 *
 *  If 1/2 of a T3 was attacking SYN packets, we'ld have a permanent queue
 *  of 70000 limbo'd calls.  Not great for a linear list but doable.  Therefore
 *  there is no hashing of this list.
 */
⟨struct Limbo 410a⟩

⟨global tcp_irtt 410b⟩

⟨enum _anon_ (kernel/network/ip/tcp.c)2 410c⟩

⟨global statnames((kernel/network/ip/tcp.c)) 411⟩

⟨struct Tcppriv 412a⟩

⟨global tcpporthogdefense 412b⟩

static  int addreseq(Fs*, Tcpctl*, Tcppriv*, Tcp*, Block*, ushort);
static  int dumpreseq(Tcpctl*);
static  void    getreseq(Tcpctl*, Tcp*, Block**, ushort*);
static  void    limbo(Conv*, uchar*, uchar*, Tcp*, int);
static  void    limborexmit(Proto*);
static  void    localclose(Conv*, char*);
static  void    procsyn(Conv*, Tcp*);
static  void    tcpacktimer(void*);
static  void    tcpiput(Proto*, Ipifc*, Block*);
static  void    tcpkeepalive(void*);
```

```
static  void    tcpoutput(Conv*);
static  void    tcprcvwin(Conv*);
static  void    tcprxmit(Conv*);
static  void    tcpsetkacounter(Tcpctl*);
static  void    tcpsetscale(Conv*, Tcpctl*, ushort, ushort);
static  void    tcpsettimer(Tcpctl*);
static  void    tcpsndsyn(Conv*, Tcpctl*);
static  void    tcpstart(Conv*, int);
static  void    tcpsynackrtt(Conv*);
static  void    tcptimeout(void*);
static  int tcptrim(Tcpctl*, Tcp*, Block**, ushort*);

⟨function tcpsetstate 412c⟩

⟨function tcpconnect 413a⟩

⟨function tcpstate 413b⟩

⟨function tcpinuse 414a⟩

⟨function tcpannounce 414b⟩

⟨function tcpclose 414c⟩

⟨function tcpkick 415a⟩

static int seq_lt(ulong, ulong);

⟨function tcprcvwin 415b⟩

⟨function tcpacktimer 416a⟩

⟨function tcpcongestion 416b⟩

⟨enum _anon_ (kernel/network/ip/tcp.c)3 417a⟩

⟨function tcpabcincr 417b⟩

⟨function tcpcreate 417c⟩

⟨function timerstate 417d⟩

⟨function tcpackproc 418⟩

⟨function tcpgo 419a⟩

⟨function tcphalt 419b⟩

⟨function backoff 419c⟩

⟨function localclose 419d⟩

⟨function tcpmtu 420a⟩

⟨function inittcpctl 420b⟩

⟨function tcpstart 421⟩

//static char*
//tcpflag(char *buf, char *e, ushort flag)
```

```
//{
//  char *p;
//
//  p = seprint(buf, e, "%d", flag>>10);    /* Head len */
//  if(flag & URG)
//      p = seprint(p, e, " URG");
//  if(flag & ACK)
//      p = seprint(p, e, " ACK");
//  if(flag & PSH)
//      p = seprint(p, e, " PSH");
//  if(flag & RST)
//      p = seprint(p, e, " RST");
//  if(flag & SYN)
//      p = seprint(p, e, " SYN");
//  if(flag & FIN)
//      p = seprint(p, e, " FIN");
//  USED(p);
//  return buf;
```
⟨*function* `htontcp6` 422⟩

⟨*function* `htontcp4` 424⟩

⟨*function* `ntohtcp6` 425⟩

⟨*function* `ntohtcp4` 426⟩

⟨*function* `tcpsndsyn` 427⟩

⟨*function* `sndrst` 428⟩

⟨*function* `tcphangup` 429⟩

⟨*function* `sndsynack` 430⟩

⟨*macro* `hashipa`*(*`(kernel/network/ip/tcp.c)`*)* 431a⟩

⟨*function* `limbo` 431b⟩

⟨*function* `limborexmit` 432⟩

⟨*function* `limborst` 433⟩

⟨*function* `initialwindow` 434a⟩

⟨*function* `tcpincoming` 434b⟩

⟨*function* `seq_within` 436⟩

⟨*function* `seq_lt` 437a⟩

⟨*function* `seq_le` 437b⟩

⟨*function* `seq_gt` 437c⟩

⟨*function* `seq_ge` 437d⟩

⟨*function* `tcpsynackrtt` 437e⟩

⟨*function* `update` 438⟩

⟨*function* tcpiput 441⟩

⟨*function* tcpoutput 449⟩

⟨*function* tcpsendka 453⟩

⟨*function* tcpsetkacounter 454a⟩

⟨*function* tcpkeepalive 454b⟩

⟨*function* tcpstartka 455a⟩

⟨*function* tcpsetchecksum 455b⟩

⟨*function* tcprxmit 455c⟩

⟨*function* tcptimeout 456⟩

⟨*function* inwindow 457a⟩

⟨*function* procsyn 457b⟩

⟨*function* dumpreseq 458a⟩

⟨*function* logreseq 458b⟩

⟨*function* addreseq 458c⟩

⟨*function* getreseq 459a⟩

⟨*function* tcptrim 459b⟩

⟨*function* tcpadvise 461⟩

⟨*function* tcpporthogdefensectl 462a⟩

⟨*function* tcpctl 462b⟩

⟨*function* tcpstats 462c⟩

⟨*function* tcpgc 462d⟩

⟨*function* tcpsettimer 463a⟩

⟨*function* tcpinit 170⟩

⟨*function* tcpsetscale 463b⟩

Uses `Limbo` 410a, `Reseq` 409a, `Tcp` 408b, `Tcp4hdr` 407c, `Tcp6hdr` 408a, `Tcpctl` 409b, `Tcppriv` 412a, and `Tcptimer` 407b.

I.9 kernel/network/ip/ ipv6

kernel/network/ip/icmp6.c

⟨*enum* _anon_ (kernel/network/ip/icmp6.c) 468⟩≡ (484b)

```
enum
{
    InMsgs6,
    InErrors6,
```

```
    OutMsgs6,
    CsumErrs6,
    LenErrs6,
    HlenErrs6,
    HoplimErrs6,
    IcmpCodeErrs6,
    TargetErrs6,
    OptlenErrs6,
    AddrmxpErrs6,
    RouterAddrErrs6,

    Nstats6,
};
```

⟨*enum* _anon_ (kernel/network/ip/icmp6.c)2 469a⟩≡ (484b)

```
enum {
    ICMP_USEAD6 = 40,
};
```

⟨*enum* _anon_ (kernel/network/ip/icmp6.c)3 469b⟩≡ (484b)

```
enum {
    Oflag   = 1<<5,
    Sflag   = 1<<6,
    Rflag   = 1<<7,
};
```

⟨*enum* _anon_ (kernel/network/ip/icmp6.c)4 469c⟩≡ (484b)

```
enum {
    /* ICMPv6 types */
    EchoReply   = 0,
    UnreachableV6   = 1,
    PacketTooBigV6  = 2,
    TimeExceedV6    = 3,
    SrcQuench   = 4,
    ParamProblemV6  = 4,
    Redirect    = 5,
    EchoRequest = 8,
    TimeExceed  = 11,
    InParmProblem   = 12,
    Timestamp   = 13,
    TimestampReply  = 14,
    InfoRequest = 15,
    InfoReply   = 16,
    AddrMaskRequest = 17,
    AddrMaskReply   = 18,
    EchoRequestV6   = 128,
    EchoReplyV6 = 129,
    RouterSolicit   = 133,
    RouterAdvert    = 134,
    NbrSolicit  = 135,
    NbrAdvert   = 136,
    RedirectV6  = 137,

    Maxtype6    = 137,
};
```

⟨*struct* IPICMP 469d⟩≡ (484b)

```
struct IPICMP {
    ICMPHDR;
    uchar   payload[];
};
```

⟨*constant* IPICMPSZ 470a⟩≡ (484b)

```
#define IPICMPSZ offsetof(IPICMP, payload[0])
```

⟨*struct* NdiscC 470b⟩≡ (484b)

```
// used though NDISCSZ macro below
struct NdiscC {
    ICMPHDR;
    uchar   target[IPaddrlen];
    uchar   payload[];
};
```

Uses IPaddrlen 20c.

⟨*constant* NDISCSZ 470c⟩≡ (484b)

```
#define NDISCSZ offsetof(NdiscC, payload[0])
```

⟨*struct* Ndpkt 470d⟩≡ (484b)

```
struct Ndpkt {
    ICMPHDR;
    uchar   target[IPaddrlen];
    uchar   otype;
    uchar   olen;       /* length in units of 8 octets(incl type, code),
                 * 1 for IEEE 802 addresses */
    uchar   lnaddr[6];  /* link-layer address */
    uchar   payload[];
};
```

Uses IPaddrlen 20c.

⟨*constant* NDPKTSZ 470e⟩≡ (484b)

```
#define NDPKTSZ offsetof(Ndpkt, payload[0])
```

⟨*struct* Icmppriv6 470f⟩≡ (484b)

```
typedef struct Icmppriv6
{
    ulong   stats[Nstats6];

    /* message counts */
    ulong   in[Maxtype6+1];
    ulong   out[Maxtype6+1];
} Icmppriv6;
```

Uses Icmppriv6 470f, Maxtype6-174 469c, and Nstats6-146 468.

⟨*struct* Icmpcb6 470g⟩≡ (484b)

```
typedef struct Icmpcb6
{
    QLock;
    uchar   headers;
} Icmpcb6;
```

Uses Icmpcb6 470g.

⟨*global* icmpnames6 470h⟩≡ (484b)

```
char *icmpnames6[Maxtype6+1] =
{
[EchoReply]     "EchoReply",
[UnreachableV6]     "UnreachableV6",
[PacketTooBigV6]    "PacketTooBigV6",
[TimeExceedV6]      "TimeExceedV6",
[SrcQuench]     "SrcQuench",
[Redirect]      "Redirect",
[EchoRequest]       "EchoRequest",
```

```
[TimeExceed]        "TimeExceed",
[InParmProblem]     "InParmProblem",
[Timestamp]     "Timestamp",
[TimestampReply]    "TimestampReply",
[InfoRequest]       "InfoRequest",
[InfoReply]     "InfoReply",
[AddrMaskRequest]   "AddrMaskRequest",
[AddrMaskReply]     "AddrMaskReply",
[EchoRequestV6]     "EchoRequestV6",
[EchoReplyV6]       "EchoReplyV6",
[RouterSolicit]     "RouterSolicit",
[RouterAdvert]      "RouterAdvert",
[NbrSolicit]        "NbrSolicit",
[NbrAdvert]     "NbrAdvert",
[RedirectV6]        "RedirectV6",
};
```

Uses `Maxtype6-174` 469c.

⟨*global* `statnames6` 471a⟩≡ (484b)

```
static char *statnames6[Nstats6] =
{
[InMsgs6]   "InMsgs",
[InErrors6] "InErrors",
[OutMsgs6]  "OutMsgs",
[CsumErrs6] "CsumErrs",
[LenErrs6]  "LenErrs",
[HlenErrs6] "HlenErrs",
[HoplimErrs6]   "HoplimErrs",
[IcmpCodeErrs6] "IcmpCodeErrs",
[TargetErrs6]   "TargetErrs",
[OptlenErrs6]   "OptlenErrs",
[AddrmxpErrs6]  "AddrmxpErrs",
[RouterAddrErrs6]   "RouterAddrErrs",
};
```

Uses `Nstats6-146` 468.

⟨*global* `unreachcode`*(*`(kernel/network/ip/icmp6.c)`*)* 471b⟩≡ (484b)

```
static char *unreachcode[] =
{
[Icmp6_no_route]    "no route to destination",
[Icmp6_ad_prohib]   "comm with destination administratively prohibited",
[Icmp6_out_src_scope]   "beyond scope of source address",
[Icmp6_adr_unreach] "address unreachable",
[Icmp6_port_unreach]    "port unreachable",
[Icmp6_gress_src_fail]  "source address failed ingress/egress policy",
[Icmp6_rej_route]   "reject route to destination",
[Icmp6_unknown]     "icmp unreachable: unknown code",
};
```

⟨*function* `icmpcreate6` 471c⟩≡ (484b)

```
static void
icmpcreate6(Conv *c)
{
    c->rq = qopen(64*1024, Qmsg, 0, c);
    c->wq = qbypass(icmpkick6, c);
}
```

Uses `icmpkick6()` 472d.

⟨*function* set_cksum 472a⟩≡ (484b)
```
static void
set_cksum(Block *bp)
{
    IPICMP *p = (IPICMP *)(bp->rp);

    hnputl(p->vcf, 0);      /* borrow IP header as pseudoheader */
    hnputs(p->ploadlen, blocklen(bp) - IP6HDR);
    p->proto = 0;
    p->ttl = ICMPv6;    /* ttl gets set later */
    hnputs(p->cksum, 0);
    hnputs(p->cksum, ptclcsum(bp, 0, blocklen(bp)));
    p->proto = ICMPv6;
}
```
Uses ICMPv6 498d, IP6HDR 498e, and ptclcsum() 90d.

⟨*function* newIPICMP 472b⟩≡ (484b)
```
static Block *
newIPICMP(int packetlen)
{
    Block *nbp;

    nbp = allocb(packetlen);
    nbp->wp += packetlen;
    memset(nbp->rp, 0, packetlen);
    return nbp;
}
```

⟨*function* icmpadvise6 472c⟩≡ (484b)
```
void
icmpadvise6(Proto *icmp, Block *bp, char *msg)
{
    ushort recid;
    Conv **c, *s;
    IPICMP *p;

    p = (IPICMP *)bp->rp;
    recid = nhgets(p->icmpid);

    for(c = icmp->conv; *c; c++) {
        s = *c;
        if(s->lport == recid && ipcmp(s->raddr, p->dst) == 0){
            qhangup(s->rq, msg);
            qhangup(s->wq, msg);
            break;
        }
    }
    freeblist(bp);
}
```
Uses ipcmp 23d.

⟨*function* icmpkick6 472d⟩≡ (484b)
```
static void
icmpkick6(void *x, Block *bp)
{
    uchar laddr[IPaddrlen], raddr[IPaddrlen];
    Conv *c = x;
    IPICMP *p;
    Icmppriv6 *ipriv = c->p->priv;
    Icmpcb6 *icb = (Icmpcb6*)c->ptcl;
```

```
    if(bp == nil)
        return;

    if(icb->headers==6) {
        /* get user specified addresses */
        bp = pullupblock(bp, ICMP_USEAD6);
        if(bp == nil)
            return;
        bp->rp += 8;
        ipmove(laddr, bp->rp);
        bp->rp += IPaddrlen;
        ipmove(raddr, bp->rp);
        bp->rp += IPaddrlen;
        bp = padblock(bp, IP6HDR);
    }

    if(blocklen(bp) < IPICMPSZ){
        freeblist(bp);
        return;
    }
    p = (IPICMP *)(bp->rp);
    if(icb->headers == 6) {
        ipmove(p->dst, raddr);
        ipmove(p->src, laddr);
    } else {
        ipmove(p->dst, c->raddr);
        ipmove(p->src, c->laddr);
        hnputs(p->icmpid, c->lport);
    }

    set_cksum(bp);
    p->vcf[0] = 0x06 << 4;
    if(p->type <= Maxtype6)
        ipriv->out[p->type]++;
    ipoput6(c->p->f, bp, 0, c->ttl, c->tos, nil);
}
```

Uses `ICMP_USEAD6-147` 469a, `IP6HDR` 498e, `IPICMPSZ-175` 470a, `IPaddrlen` 20c, `Maxtype6-174` 469c, `ipmove` 23c, `ipoput6()` 486e, and `set_cksum()` 472a.

⟨*function* `icmpctl6` 473a⟩≡ (484b)

```
char*
icmpctl6(Conv *c, char **argv, int argc)
{
    Icmpcb6 *icb;

    icb = (Icmpcb6*) c->ptcl;
    if(argc==1 && strcmp(argv[0], "headers")==0) {
        icb->headers = 6;
        return nil;
    }
    return "unknown control request";
}
```

⟨*function* `goticmpkt6` 473b⟩≡ (484b)

```
static void
goticmpkt6(Proto *icmp, Block *bp, int muxkey)
{
    ushort recid;
    uchar *addr;
```

```
    Conv **c, *s;
    IPICMP *p = (IPICMP *)bp->rp;

    if(muxkey == 0) {
        recid = nhgets(p->icmpid);
        addr = p->src;
    } else {
        recid = muxkey;
        addr = p->dst;
    }

    for(c = icmp->conv; *c; c++){
        s = *c;
        if(s->lport == recid && ipcmp(s->raddr, addr) == 0){
            bp = concatblock(bp);
            if(bp != nil)
                qpass(s->rq, bp);
            return;
        }
    }

    freeblist(bp);
}
```

Uses ipcmp 23d.

⟨*function* mkechoreply6 474a⟩≡ (484b)

```
static Block *
mkechoreply6(Block *bp, Ipifc *ifc)
{
    uchar addr[IPaddrlen];
    IPICMP *p = (IPICMP *)(bp->rp);

    ipmove(addr, p->src);
    if(!isv6mcast(p->dst))
        ipmove(p->src, p->dst);
    else if (!ipv6anylocal(ifc, p->src))
        return nil;
    ipmove(p->dst, addr);
    p->type = EchoReplyV6;
    set_cksum(bp);
    return bp;
}
```

Uses EchoReplyV6-168 469c, IPaddrlen 20c, ipmove 23c, ipv6anylocal() 512b, isv6mcast 497b, and set_cksum() 472a.

⟨*function* icmpns 474b⟩≡ (484b)

```
/*
 * sends out an ICMPv6 neighbor solicitation
 *  suni == SRC_UNSPEC or SRC_UNI,
 *  tuni == TARG_MULTI => multicast for address resolution,
 *  and tuni == TARG_UNI => neighbor reachability.
 */
extern void
icmpns(Fs *f, uchar* src, int suni, uchar* targ, int tuni, uchar* mac)
{
    Block *nbp;
    Ndpkt *np;
    Proto *icmp = f->t2p[ICMPv6];
    Icmppriv6 *ipriv = icmp->priv;

    nbp = newIPICMP(NDPKTSZ);
```

```
    np = (Ndpkt*) nbp->rp;

    if(suni == SRC_UNSPEC)
        memmove(np->src, v6Unspecified, IPaddrlen);
    else
        memmove(np->src, src, IPaddrlen);

    if(tuni == TARG_UNI)
        memmove(np->dst, targ, IPaddrlen);
    else
        ipv62smcast(np->dst, targ);

    np->type = NbrSolicit;
    np->code = 0;
    memmove(np->target, targ, IPaddrlen);
    if(suni != SRC_UNSPEC) {
        np->otype = SRC_LLADDR;
        np->olen = 1;       /* 1+1+6 = 8 = 1 8-octet */
        memmove(np->lnaddr, mac, sizeof(np->lnaddr));
    } else
        nbp->wp -= NDPKTSZ - NDISCSZ;

    set_cksum(nbp);
    np = (Ndpkt*)nbp->rp;
    np->ttl = HOP_LIMIT;
    np->vcf[0] = 0x06 << 4;
    ipriv->out[NbrSolicit]++;
    netlog(f, Logicmp, "sending neighbor solicitation %I\n", targ);
    ipoput6(f, nbp, 0, MAXTTL, DFLTTOS, nil);
}
```

Uses `DFLTTOS` 232b, `HOP_LIMIT` 498e, `ICMPv6` 498d, `IPaddrlen` 20c, `Logicmp` 233d, `MAXTTL` 81g, `NDISCSZ-176` 470c, `NDPKTSZ-177` 470e, `NbrSolicit-171` 469c, `SRC_LLADDR` 498e, `SRC_UNSPEC` 498e, `TARG_UNI` 498e, `ipoput6()` 486e, `ipv62smcast()` 372e, `netlog()` 389a, `newIPICMP()` 472b, `set_cksum()` 472a, and `v6Unspecified` 371a.

⟨*function* `icmpna` 475⟩≡ (484b)

```
/*
 * sends out an ICMPv6 neighbor advertisement. pktflags == RSO flags.
 */
extern void
icmpna(Fs *f, uchar* src, uchar* dst, uchar* targ, uchar* mac, uchar flags)
{
    Block *nbp;
    Ndpkt *np;
    Proto *icmp = f->t2p[ICMPv6];
    Icmppriv6 *ipriv = icmp->priv;

    nbp = newIPICMP(NDPKTSZ);
    np = (Ndpkt*)nbp->rp;

    memmove(np->src, src, IPaddrlen);
    memmove(np->dst, dst, IPaddrlen);

    np->type = NbrAdvert;
    np->code = 0;
    np->icmpid[0] = flags;
    memmove(np->target, targ, IPaddrlen);

    np->otype = TARGET_LLADDR;
    np->olen = 1;
    memmove(np->lnaddr, mac, sizeof(np->lnaddr));
```

```
    set_cksum(nbp);
    np = (Ndpkt*) nbp->rp;
    np->ttl = HOP_LIMIT;
    np->vcf[0] = 0x06 << 4;
    ipriv->out[NbrAdvert]++;
    netlog(f, Logicmp, "sending neighbor advertisement %I\n", src);
    ipoput6(f, nbp, 0, MAXTTL, DFLTTOS, nil);
}
```

Uses DFLTTOS 232b, HOP_LIMIT 498e, ICMPv6 498d, IPaddrlen 20c, Logicmp 233d, MAXTTL 81g, NDPKTSZ-177 470e, NbrAdvert-172 469c, TARGET_LLADDR 498e, ipoput6() 486e, netlog() 389a, newIPICMP() 472b, and set_cksum() 472a.

⟨*function* icmphostunr 476⟩≡ (484b)

```
extern void
icmphostunr(Fs *f, Ipifc *ifc, Block *bp, int code, int free)
{
    int osz = BLEN(bp);
    int sz = MIN(IPICMPSZ + osz, v6MINTU);
    Block *nbp;
    IPICMP *np;
    Ip6hdr *p;
    Proto *icmp = f->t2p[ICMPv6];
    Icmppriv6 *ipriv = icmp->priv;

    p = (Ip6hdr *)bp->rp;

    if(isv6mcast(p->src))
        goto clean;

    nbp = newIPICMP(sz);
    np = (IPICMP *)nbp->rp;

    rlock(ifc);
    if(ipv6anylocal(ifc, np->src))
        netlog(f, Logicmp, "send icmphostunr -> src %I dst %I\n",
            p->src, p->dst);
    else {
        netlog(f, Logicmp, "icmphostunr fail -> src %I dst %I\n",
            p->src, p->dst);
        freeblist(nbp);
        if(free)
            goto clean;
        else
            return;
    }

    memmove(np->dst, p->src, IPaddrlen);
    np->type = UnreachableV6;
    np->code = code;
    memmove(nbp->rp + IPICMPSZ, bp->rp, sz - IPICMPSZ);
    set_cksum(nbp);
    np->ttl = HOP_LIMIT;
    np->vcf[0] = 0x06 << 4;
    ipriv->out[UnreachableV6]++;

    if(free)
        ipiput6(f, ifc, nbp);
    else {
        ipoput6(f, nbp, 0, MAXTTL, DFLTTOS, nil);
        return;
```

```
    }

clean:
    runlock(ifc);
    freeblist(bp);
}
```

Uses DFLTTOS 232b, HOP_LIMIT 498e, ICMPv6 498d, IPICMPSZ-175 470a, IPaddrlen 20c, Logicmp 233d, MAXTTL 81g, UnreachableV6-152 469c, ipiput6() 490, ipoput6() 486e, ipv6anylocal() 512b, isv6mcast 497b, netlog() 389a, newIPICMP() 472b, set_cksum() 472a, and v6MINTU 498e.

⟨*function* icmpttlexceeded6 477a⟩≡ (484b)

```
extern void
icmpttlexceeded6(Fs *f, Ipifc *ifc, Block *bp)
{
    int osz = BLEN(bp);
    int sz = MIN(IPICMPSZ + osz, v6MINTU);
    Block *nbp;
    IPICMP *np;
    Ip6hdr *p;
    Proto *icmp = f->t2p[ICMPv6];
    Icmppriv6 *ipriv = icmp->priv;

    p = (Ip6hdr *)bp->rp;

    if(isv6mcast(p->src))
        return;

    nbp = newIPICMP(sz);
    np = (IPICMP *) nbp->rp;

    if(ipv6anylocal(ifc, np->src))
        netlog(f, Logicmp, "send icmpttlexceeded6 -> src %I dst %I\n",
            p->src, p->dst);
    else {
        netlog(f, Logicmp, "icmpttlexceeded6 fail -> src %I dst %I\n",
            p->src, p->dst);
        return;
    }

    memmove(np->dst, p->src, IPaddrlen);
    np->type = TimeExceedV6;
    np->code = 0;
    memmove(nbp->rp + IPICMPSZ, bp->rp, sz - IPICMPSZ);
    set_cksum(nbp);
    np->ttl = HOP_LIMIT;
    np->vcf[0] = 0x06 << 4;
    ipriv->out[TimeExceedV6]++;
    ipoput6(f, nbp, 0, MAXTTL, DFLTTOS, nil);
}
```

Uses DFLTTOS 232b, HOP_LIMIT 498e, ICMPv6 498d, IPICMPSZ-175 470a, IPaddrlen 20c, Logicmp 233d, MAXTTL 81g, TimeExceedV6-154 469c, ipoput6() 486e, ipv6anylocal() 512b, isv6mcast 497b, netlog() 389a, newIPICMP() 472b, set_cksum() 472a, and v6MINTU 498e.

⟨*function* icmppkttoobig6 477b⟩≡ (484b)

```
extern void
icmppkttoobig6(Fs *f, Ipifc *ifc, Block *bp)
{
    int osz = BLEN(bp);
    int sz = MIN(IPICMPSZ + osz, v6MINTU);
    Block *nbp;
```

```
    IPICMP *np;
    Ip6hdr *p;
    Proto *icmp = f->t2p[ICMPv6];
    Icmppriv6 *ipriv = icmp->priv;

    p = (Ip6hdr *)bp->rp;

    if(isv6mcast(p->src))
        return;

    nbp = newIPICMP(sz);
    np = (IPICMP *)nbp->rp;

    if(ipv6anylocal(ifc, np->src))
        netlog(f, Logicmp, "send icmppkttoobig6 -> src %I dst %I\n",
            p->src, p->dst);
    else {
        netlog(f, Logicmp, "icmppkttoobig6 fail -> src %I dst %I\n",
            p->src, p->dst);
        return;
    }

    memmove(np->dst, p->src, IPaddrlen);
    np->type = PacketTooBigV6;
    np->code = 0;
    hnputl(np->icmpid, ifc->maxtu - ifc->m->hsize);
    memmove(nbp->rp + IPICMPSZ, bp->rp, sz - IPICMPSZ);
    set_cksum(nbp);
    np->ttl = HOP_LIMIT;
    np->vcf[0] = 0x06 << 4;
    ipriv->out[PacketTooBigV6]++;
    ipoput6(f, nbp, 0, MAXTTL, DFLTTOS, nil);
}
```

Uses DFLTTOS 232b, HOP_LIMIT 498e, ICMPv6 498d, IPICMPSZ-175 470a, IPaddrlen 20c, Logicmp 233d, MAXTTL 81g, PacketTooBigV6-153 469c, ipoput6() 486e, ipv6anylocal() 512b, isv6mcast 497b, netlog() 389a, newIPICMP() 472b, set_cksum() 472a, and v6MINTU 498e.

⟨*function* valid 478⟩≡ (484b)

```
/*
 * RFC 2461, pages 39-40, pages 57-58.
 */
static int
valid(Proto *icmp, Ipifc *ifc, Block *bp, Icmppriv6 *ipriv)
{
    int sz, osz, unsp, n, ttl, iplen;
    int pktsz = BLEN(bp);
    uchar *packet = bp->rp;
    IPICMP *p = (IPICMP *) packet;
    Ndpkt *np;

    USED(ifc);
    n = blocklen(bp);
    if(n < IPICMPSZ) {
        ipriv->stats[HlenErrs6]++;
        netlog(icmp->f, Logicmp, "icmp hlen %d\n", n);
        goto err;
    }

    iplen = nhgets(p->ploadlen);
    if(iplen > n - IP6HDR) {
```

```
    ipriv->stats[LenErrs6]++;
    netlog(icmp->f, Logicmp, "icmp length %d\n", iplen);
    goto err;
}

/* Rather than construct explicit pseudoheader, overwrite IPv6 header */
if(p->proto != ICMPv6) {
    /* This code assumes no extension headers!!! */
    netlog(icmp->f, Logicmp, "icmp error: extension header\n");
    goto err;
}
memset(packet, 0, 4);
ttl = p->ttl;
p->ttl = p->proto;
p->proto = 0;
if(ptclcsum(bp, 0, iplen + IP6HDR)) {
    ipriv->stats[CsumErrs6]++;
    netlog(icmp->f, Logicmp, "icmp checksum error\n");
    goto err;
}
p->proto = p->ttl;
p->ttl = ttl;

/* additional tests for some pkt types */
if (p->type == NbrSolicit   || p->type == NbrAdvert ||
    p->type == RouterAdvert || p->type == RouterSolicit ||
    p->type == RedirectV6) {
    if(p->ttl != HOP_LIMIT) {
        ipriv->stats[HoplimErrs6]++;
        goto err;
    }
    if(p->code != 0) {
        ipriv->stats[IcmpCodeErrs6]++;
        goto err;
    }

    switch (p->type) {
    case NbrSolicit:
    case NbrAdvert:
        np = (Ndpkt*) p;
        if(isv6mcast(np->target)) {
            ipriv->stats[TargetErrs6]++;
            goto err;
        }
        if(optexsts(np) && np->olen == 0) {
            ipriv->stats[OptlenErrs6]++;
            goto err;
        }

        if (p->type == NbrSolicit &&
            ipcmp(np->src, v6Unspecified) == 0)
            if(!issmcast(np->dst) || optexsts(np)) {
                ipriv->stats[AddrmxpErrs6]++;
                goto err;
            }

        if(p->type == NbrAdvert)
            if(isv6mcast(np->dst) &&
                (nhgets(np->icmpid) & Sflag)){
                ipriv->stats[AddrmxpErrs6]++;
```

```
                goto err;
            }
        break;

    case RouterAdvert:
        if(pktsz - IP6HDR < 16) {
            ipriv->stats[HlenErrs6]++;
            goto err;
        }
        if(!islinklocal(p->src)) {
            ipriv->stats[RouterAddrErrs6]++;
            goto err;
        }
        sz = IPICMPSZ + 8;
        while (sz+1 < pktsz) {
            osz = packet[sz+1];
            if(osz <= 0) {
                ipriv->stats[OptlenErrs6]++;
                goto err;
            }
            sz += 8*osz;
        }
        break;

    case RouterSolicit:
        if(pktsz - IP6HDR < 8) {
            ipriv->stats[HlenErrs6]++;
            goto err;
        }
        unsp = (ipcmp(p->src, v6Unspecified) == 0);
        sz = IPICMPSZ + 8;
        while (sz+1 < pktsz) {
            osz = packet[sz+1];
            if(osz <= 0 ||
                (unsp && packet[sz] == SRC_LLADDR)) {
                ipriv->stats[OptlenErrs6]++;
                goto err;
            }
            sz += 8*osz;
        }
        break;

    case RedirectV6:
        /* to be filled in */
        break;

    default:
        goto err;
    }
  }
  return 1;
err:
  ipriv->stats[InErrors6]++;
  return 0;
}
```

Uses AddrmxpErrs6-144 468, CsumErrs6-137 468, HOP_LIMIT 498e, HlenErrs6-139 468, HoplimErrs6-140 468, ICMPv6 498d, IP6HDR 498e, IPICMPSZ-175 470a, IcmpCodeErrs6-141 468, InErrors6-135 468, LenErrs6-138 468, Logicmp 233d, NbrAdvert-172 469c, NbrSolicit-171 469c, OptlenErrs6-143 468, RedirectV6-173 469c, RouterAddrErrs6-145 468, RouterAdvert-170 469c, RouterSolicit-169 469c, SRC_LLADDR 498e, Sflag-149 469b, TargetErrs6-142 468, ipcmp 23d,

islinklocal 498a, issmcast 498c, isv6mcast 497b, netlog() 389a, optexsts 498b, ptclcsum() 90d, and v6Unspecified 371a.

⟨*function* targettype 481a⟩≡ (484b)

```
static int
targettype(Fs *f, Ipifc *ifc, uchar *target)
{
    Iplifc *lifc;
    int t;

    rlock(ifc);
    if(ipproxyifc(f, ifc, target)) {
        runlock(ifc);
        return Tuniproxy;
    }

    for(lifc = ifc->lifc; lifc; lifc = lifc->next)
        if(ipcmp(lifc->local, target) == 0) {
            t = (lifc->tentative)? Tunitent: Tunirany;
            runlock(ifc);
            return t;
        }

    runlock(ifc);
    return 0;
}
```

Uses Tuniproxy 498e, Tunirany 498e, Tunitent 498e, ipcmp 23d, and ipproxyifc() 381b.

⟨*function* icmpiput6 481b⟩≡ (484b)

```
static void
icmpiput6(Proto *icmp, Ipifc *ipifc, Block *bp)
{
    int refresh = 1;
    char *msg, m2[128];
    uchar pktflags;
    uchar *packet = bp->rp;
    uchar lsrc[IPaddrlen];
    Block *r;
    IPICMP *p = (IPICMP *)packet;
    Icmppriv6 *ipriv = icmp->priv;
    Iplifc *lifc;
    Ndpkt* np;
    Proto *pr;

    if(!valid(icmp, ipifc, bp, ipriv) || p->type > Maxtype6)
        goto raise;

    ipriv->in[p->type]++;

    switch(p->type) {
    case EchoRequestV6:
        r = mkechoreply6(bp, ipifc);
        if(r == nil)
            goto raise;
        ipriv->out[EchoReply]++;
        ipoput6(icmp->f, r, 0, MAXTTL, DFLTTOS, nil);
        break;

    case UnreachableV6:
        if(p->code >= nelem(unreachcode))
            msg = unreachcode[Icmp6_unknown];
```

```
		else
			msg = unreachcode[p->code];

		bp->rp += IPICMPSZ;
		if(blocklen(bp) < 8){
			ipriv->stats[LenErrs6]++;
			goto raise;
		}
		p = (IPICMP *)bp->rp;
		pr = Fsrcvpcolx(icmp->f, p->proto);
		if(pr != nil && pr->advise != nil) {
			(*pr->advise)(pr, bp, msg);
			return;
		}

		bp->rp -= IPICMPSZ;
		goticmpkt6(icmp, bp, 0);
		break;

	case TimeExceedV6:
		if(p->code == 0){
			snprint(m2, sizeof m2, "ttl exceeded at %I", p->src);

			bp->rp += IPICMPSZ;
			if(blocklen(bp) < 8){
				ipriv->stats[LenErrs6]++;
				goto raise;
			}
			p = (IPICMP *)bp->rp;
			pr = Fsrcvpcolx(icmp->f, p->proto);
			if(pr && pr->advise) {
				(*pr->advise)(pr, bp, m2);
				return;
			}
			bp->rp -= IPICMPSZ;
		}

		goticmpkt6(icmp, bp, 0);
		break;

	case RouterAdvert:
	case RouterSolicit:
		/* using lsrc as a temp, munge hdr for goticmp6 */
		if (0) {
			memmove(lsrc, p->src, IPaddrlen);
			memmove(p->src, p->dst, IPaddrlen);
			memmove(p->dst, lsrc, IPaddrlen);
		}
		goticmpkt6(icmp, bp, p->type);
		break;

	case NbrSolicit:
		np = (Ndpkt*) p;
		pktflags = 0;
		switch (targettype(icmp->f, ipifc, np->target)) {
		case Tunirany:
			pktflags |= Oflag;
			/* fall through */

		case Tuniproxy:
```

```
            if(ipcmp(np->src, v6Unspecified) != 0) {
                arpenter(icmp->f, V6, np->src, np->lnaddr,
                    8*np->olen-2, 0);
                pktflags |= Sflag;
            }
            if(ipv6local(ipifc, lsrc))
                icmpna(icmp->f, lsrc,
                    (ipcmp(np->src, v6Unspecified) == 0?
                        v6allnodesL: np->src),
                    np->target, ipifc->mac, pktflags);
            else
                freeblist(bp);
            break;

        case Tunitent:
            /* not clear what needs to be done. send up
             * an icmp mesg saying don't use this address? */
        default:
            freeblist(bp);
        }
        break;

    case NbrAdvert:
        np = (Ndpkt*) p;

        /*
         * if the target address matches one of the local interface
         * addresses and the local interface address has tentative bit
         * set, insert into ARP table. this is so the duplicate address
         * detection part of ipconfig can discover duplication through
         * the arp table.
         */
        lifc = iplocalonifc(ipifc, np->target);
        if(lifc && lifc->tentative)
            refresh = 0;
        arpenter(icmp->f, V6, np->target, np->lnaddr, 8*np->olen-2,
            refresh);
        freeblist(bp);
        break;

    case PacketTooBigV6:
    default:
        goticmpkt6(icmp, bp, 0);
        break;
    }
    return;
raise:
    freeblist(bp);
}
```

Uses DFLTTOS 232b, EchoReply-151 469c, EchoRequestV6-167 469c, Fsrcvpcolx() 331a, IPICMPSZ-175 470a, IPaddrlen 20c, Icmp6_unknown 498e, LenErrs6-138 468, MAXTTL 81g, Maxtype6-174 469c, NbrAdvert-172 469c, NbrSolicit-171 469c, Oflag-148 469b, PacketTooBigV6-153 469c, RouterAdvert-170 469c, RouterSolicit-169 469c, Sflag-149 469b, TimeExceedV6-154 469c, Tuniproxy 498e, Tunirany 498e, Tunitent 498e, UnreachableV6-152 469c, V6 21g, arpenter() 338, goticmpkt6() 473b, icmpna() 475, ipcmp 23d, iplocalonifc() 381a, ipoput6() 486e, ipv6local() 512a, mkechoreply6() 474a, targettype() 481a, unreachcode-179 471b, v6Unspecified 371a, v6allnodesL 372a, and valid() 478.

⟨*function* icmpstats6 483⟩≡ (484b)

```
int
icmpstats6(Proto *icmp6, char *buf, int len)
{
```

```
    Icmppriv6 *priv;
    char *p, *e;
    int i;

    priv = icmp6->priv;
    p = buf;
    e = p+len;
    for(i = 0; i < Nstats6; i++)
        p = seprint(p, e, "%s: %lud\n", statnames6[i], priv->stats[i]);
    for(i = 0; i <= Maxtype6; i++)
        if(icmpnames6[i])
            p = seprint(p, e, "%s: %lud %lud\n", icmpnames6[i],
                priv->in[i], priv->out[i]);
/*      else
            p = seprint(p, e, "%d: %lud %lud\n", i, priv->in[i],
                priv->out[i]);
 */
    return p - buf;
}
```

Uses `Maxtype6-174` 469c, `Nstats6-146` 468, `icmpnames6` 470h, and `statnames6-178` 471a.

⟨*function* `icmp6init` 484a⟩≡ (484b)

```
void
icmp6init(Fs *fs)
{
    Proto *icmp6 = smalloc(sizeof(Proto));

    icmp6->priv = smalloc(sizeof(Icmppriv6));
    icmp6->name = "icmpv6";
    icmp6->connect = icmpconnect;
    icmp6->announce = icmpannounce;
    icmp6->state = icmpstate;
    icmp6->create = icmpcreate6;
    icmp6->close = icmpclose;
    icmp6->rcv = icmpiput6;
    icmp6->stats = icmpstats6;
    icmp6->ctl = icmpctl6;
    icmp6->advise = icmpadvise6;
    icmp6->gc = nil;
    icmp6->ipproto = ICMPv6;
    icmp6->nc = 16;
    icmp6->ptclsize = sizeof(Icmpcb6);

    Fsproto(fs, icmp6);
}
```

Uses `Fsproto()` 68a, `ICMPv6` 498d, `icmpadvise6()` 472c, `icmpannounce()` 345c, `icmpclose()` 345d, `icmpconnect()` 345a, `icmpcreate6()` 471c, `icmpctl6()` 473a, `icmpiput6()` 481b, `icmpstate()` 345b, and `icmpstats6()` 483.

⟨`kernel/network/ip/icmp6.c` 484b⟩≡

```
/*
 * Internet Control Message Protocol for IPv6
 */
#include "u.h"
#include "../port/lib.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "../port/error.h"
#include "ip.h"
#include "ipv6.h"
```

```
⟨enum _anon_ (kernel/network/ip/icmp6.c) 468⟩

⟨enum _anon_ (kernel/network/ip/icmp6.c)2 469a⟩

⟨enum _anon_ (kernel/network/ip/icmp6.c)3 469b⟩

⟨enum _anon_ (kernel/network/ip/icmp6.c)4 469c⟩

/* on-the-wire packet formats */
typedef struct IPICMP IPICMP;
typedef struct Ndpkt Ndpkt;
typedef struct NdiscC NdiscC;

/* we do this to avoid possible struct padding  */
#define ICMPHDR \
    IPV6HDR; \
    uchar   type; \
    uchar   code; \
    uchar   cksum[2]; \
    uchar   icmpid[2]; \
    uchar   seq[2]

⟨struct IPICMP 469d⟩

⟨constant IPICMPSZ 470a⟩

⟨struct NdiscC 470b⟩

⟨constant NDISCSZ 470c⟩

⟨struct Ndpkt 470d⟩

⟨constant NDPKTSZ 470e⟩

⟨struct Icmppriv6 470f⟩

⟨struct Icmpcb6 470g⟩

⟨global icmpnames6 470h⟩

⟨global statnames6 471a⟩

⟨global unreachcode((kernel/network/ip/icmp6.c)) 471b⟩

static void icmpkick6(void *x, Block *bp);

⟨function icmpcreate6 471c⟩

⟨function set_cksum 472a⟩

⟨function newIPICMP 472b⟩

⟨function icmpadvise6 472c⟩

⟨function icmpkick6 472d⟩

⟨function icmpctl6 473a⟩

⟨function goticmpkt6 473b⟩
```

```
⟨function mkechoreply6 474a⟩

⟨function icmpns 474b⟩

⟨function icmpna 475⟩

⟨function icmphostunr 476⟩

⟨function icmpttlexceeded6 477a⟩

⟨function icmppkttoobig6 477b⟩

⟨function valid 478⟩

⟨function targettype 481a⟩

⟨function icmpiput6 481b⟩

⟨function icmpstats6 483⟩


/* import from icmp.c */
extern int  icmpstate(Conv *c, char *state, int n);
extern char*    icmpannounce(Conv *c, char **argv, int argc);
extern char*    icmpconnect(Conv *c, char **argv, int argc);
extern void icmpclose(Conv *c);

⟨function icmp6init 484a⟩
```

Uses IPICMP 469d and Ndpkt 470d.

kernel/network/ip/ipv6.c

⟨*enum* _anon_ (kernel/network/ip/ipv6.c) 486a⟩≡ (497a)

```
enum
{
    IP6FHDR     = 8,        /* sizeof(Fraghdr6) */
};
```

⟨*macro* IPV6CLASS 486b⟩≡ (497a)

```
#define IPV6CLASS(hdr)  (((hdr)->vcf[0]&0x0F)<<2 | ((hdr)->vcf[1]&0xF0)>>2)
```

⟨*macro* BLKIPVER*(*(kernel/network/ip/ipv6.c)*)* 486c⟩≡ (497a)

```
#define BLKIPVER(xp)    (((Ip6hdr*)((xp)->rp))->vcf[0] & 0xF0)
```

⟨*macro* BKFG*(*(kernel/network/ip/ipv6.c)*)* 486d⟩≡ (497a)

```
/*
 * This sleazy macro is stolen shamelessly from ip.c, see comment there.
 */
#define BKFG(xp)    ((Ipfrag*)((xp)->base))
```

⟨*function* ipoput6 486e⟩≡ (497a)

```
int
ipoput6(Fs *f, Block *bp, int gating, int ttl, int tos, Conv *c)
{
    int medialen, len, chunk, uflen, flen, seglen, lid, offset, fragoff;
    int morefrags, blklen, rv = 0, tentative;
    uchar *gate, nexthdr;
    Block *xp, *nb;
```

```
	Fraghdr6 fraghdr;
	IP *ip;
	Ip6hdr *eh;
	Ipifc *ifc;
	Route *r, *sr;

	ip = f->ip;

	/* Fill out the ip header */
	eh = (Ip6hdr*)(bp->rp);

	ip->stats[OutRequests]++;

	/* Number of uchars in data and ip header to write */
	len = blocklen(bp);

	tentative = iptentative(f, eh->src);
	if(tentative){
		netlog(f, Logip, "reject tx of packet with tentative src address %I\n",
			eh->src);
		goto free;
	}

	if(gating){
		chunk = nhgets(eh->ploadlen);
		if(chunk > len){
			ip->stats[OutDiscards]++;
			netlog(f, Logip, "short gated packet\n");
			goto free;
		}
		if(chunk + IP6HDR < len)
			len = chunk + IP6HDR;
	}

	if(len >= IP_MAX){
		ip->stats[OutDiscards]++;
		netlog(f, Logip, "exceeded ip max size %I\n", eh->dst);
		goto free;
	}

	r = v6lookup(f, eh->dst, c);
	if(r == nil){
//		print("no route for %I, src %I free\n", eh->dst, eh->src);
		ip->stats[OutNoRoutes]++;
		netlog(f, Logip, "no interface %I\n", eh->dst);
		rv = -1;
		goto free;
	}

	ifc = r->ifc;
	if(r->type & (Rifc|Runi))
		gate = eh->dst;
	else if(r->type & (Rbcast|Rmulti)) {
		gate = eh->dst;
		sr = v6lookup(f, eh->src, nil);
		if(sr && (sr->type & Runi))
			ifc = sr->ifc;
	}
	else
		gate = r->v6.gate;
```

```
if(!gating)
    eh->vcf[0] = IP_VER6;
eh->ttl = ttl;
if(!gating) {
    eh->vcf[0] |= tos >> 4;
    eh->vcf[1]  = tos << 4;
}

if(!canrlock(ifc))
    goto free;

if(waserror()){
    runlock(ifc);
    nexterror();
}

if(ifc->m == nil)
    goto raise;

/* If we dont need to fragment just send it */
medialen = ifc->maxtu - ifc->m->hsize;
if(len <= medialen) {
    hnputs(eh->ploadlen, len - IP6HDR);
    ifc->m->bwrite(ifc, bp, V6, gate);
    runlock(ifc);
    poperror();
    return 0;
}

if(gating && ifc->reassemble <= 0) {
    /*
     * v6 intermediate nodes are not supposed to fragment pkts;
     * we fragment if ifc->reassemble is turned on; an exception
     * needed for nat.
     */
    ip->stats[OutDiscards]++;
    icmppkttoobig6(f, ifc, bp);
    netlog(f, Logip, "%I: gated pkts not fragmented\n", eh->dst);
    goto raise;
}

/* start v6 fragmentation */
uflen = unfraglen(bp, &nexthdr, 1);
if(uflen > medialen) {
    ip->stats[FragFails]++;
    ip->stats[OutDiscards]++;
    netlog(f, Logip, "%I: unfragmentable part too big\n", eh->dst);
    goto raise;
}

flen = len - uflen;
seglen = (medialen - (uflen + IP6FHDR)) & ~7;
if(seglen < 8) {
    ip->stats[FragFails]++;
    ip->stats[OutDiscards]++;
    netlog(f, Logip, "%I: seglen < 8\n", eh->dst);
    goto raise;
}
```

```
lid = incref(&ip->id6);
fraghdr.nexthdr = nexthdr;
fraghdr.res = 0;
hnputl(fraghdr.id, lid);

xp = bp;
offset = uflen;
while (xp && offset && offset >= BLEN(xp)) {
    offset -= BLEN(xp);
    xp = xp->next;
}
xp->rp += offset;

fragoff = 0;
morefrags = 1;

for(; fragoff < flen; fragoff += seglen) {
    nb = allocb(uflen + IP6FHDR + seglen);

    if(fragoff + seglen >= flen) {
        seglen = flen - fragoff;
        morefrags = 0;
    }

    hnputs(eh->ploadlen, seglen+IP6FHDR);
    memmove(nb->wp, eh, uflen);
    nb->wp += uflen;

    hnputs(fraghdr.offsetRM, fragoff); /* last 3 bits must be 0 */
    fraghdr.offsetRM[1] |= morefrags;
    memmove(nb->wp, &fraghdr, IP6FHDR);
    nb->wp += IP6FHDR;

    /* Copy data */
    chunk = seglen;
    while (chunk) {
        if(!xp) {
            ip->stats[OutDiscards]++;
            ip->stats[FragFails]++;
            freeblist(nb);
            netlog(f, Logip, "!xp: chunk in v6%d\n", chunk);
            goto raise;
        }
        blklen = chunk;
        if(BLEN(xp) < chunk)
            blklen = BLEN(xp);
        memmove(nb->wp, xp->rp, blklen);

        nb->wp += blklen;
        xp->rp += blklen;
        chunk -= blklen;
        if(xp->rp == xp->wp)
            xp = xp->next;
    }

    ifc->m->bwrite(ifc, nb, V6, gate);
    ip->stats[FragCreates]++;
}
ip->stats[FragOKs]++;
```

```
raise:
    runlock(ifc);
    poperror();
free:
    freeblist(bp);
    return rv;
}
```

Uses FragCreates 31a, FragFails 31a, FragOKs 31a, IP6FHDR-259 486a, IP6HDR 498e, IP_MAX 97a, IP_VER6 232b, Logip 233d, OutDiscards 31a, OutNoRoutes 31a, OutRequests 31a, Rbcast 38e, Rifc 38e, Rmulti 38e, Runi 38e, V6 21g, icmppkttoobig6() 477b, iptentative() 378d, netlog() 389a, unfraglen() 493b, and v6lookup() 508b.

⟨*function* ipiput6 490⟩≡ (497a)

```
void
ipiput6(Fs *f, Ipifc *ifc, Block *bp)
{
    int hl, hop, tos, notforme, tentative;
    uchar proto;
    uchar v6dst[IPaddrlen];
    IP *ip;
    Ip6hdr *h;
    Proto *p;
    Route *r, *sr;

    ip = f->ip;
    ip->stats[InReceives]++;

    /*
     *  Ensure we have all the header info in the first
     *  block.  Make life easier for other protocols by
     *  collecting up to the first 64 bytes in the first block.
     */
    if(BLEN(bp) < 64) {
        hl = blocklen(bp);
        if(hl < IP6HDR)
            hl = IP6HDR;
        if(hl > 64)
            hl = 64;
        bp = pullupblock(bp, hl);
        if(bp == nil)
            return;
    }

    h = (Ip6hdr *)bp->rp;

    memmove(&v6dst[0], &h->dst[0], IPaddrlen);
    notforme = ipforme(f, v6dst) == 0;
    tentative = iptentative(f, v6dst);

    if(tentative && h->proto != ICMPv6) {
        print("ipv6 tentative addr %I, drop\n", v6dst);
        freeblist(bp);
        return;
    }

    /* Check header version */
    if(BLKIPVER(bp) != IP_VER6) {
        ip->stats[InHdrErrors]++;
        netlog(f, Logip, "ip: bad version %ux\n", (h->vcf[0]&0xF0)>>2);
        freeblist(bp);
        return;
```

```
}

/* route */
if(notforme) {
    if(!ip->iprouting){
        freeblist(bp);
        return;
    }

    /* don't forward to link-local destinations */
    if(islinklocal(h->dst) ||
       (isv6mcast(h->dst) && (h->dst[1]&0xF) <= Link_local_scop)){
        ip->stats[OutDiscards]++;
        freeblist(bp);
        return;
    }

    /* don't forward to source's network */
    sr = v6lookup(f, h->src, nil);
    r  = v6lookup(f, h->dst, nil);

    if(r == nil || sr == r){
        ip->stats[OutDiscards]++;
        freeblist(bp);
        return;
    }

    /* don't forward if packet has timed out */
    hop = h->ttl;
    if(hop < 1) {
        ip->stats[InHdrErrors]++;
        icmpttlexceeded6(f, ifc, bp);
        freeblist(bp);
        return;
    }

    /* process headers & reassemble if the interface expects it */
    bp = procxtns(ip, bp, r->ifc->reassemble);
    if(bp == nil)
        return;

    ip->stats[ForwDatagrams]++;
    h = (Ip6hdr *)bp->rp;
    tos = IPV6CLASS(h);
    hop = h->ttl;
    ipoput6(f, bp, 1, hop-1, tos, nil);
    return;
}

/* reassemble & process headers if needed */
bp = procxtns(ip, bp, 1);
if(bp == nil)
    return;

h = (Ip6hdr *) (bp->rp);
proto = h->proto;
p = Fsrcvpcol(f, proto);
if(p && p->rcv) {
    ip->stats[InDelivers]++;
    (*p->rcv)(p, ifc, bp);
```

```
        return;
    }

    ip->stats[InDiscards]++;
    ip->stats[InUnknownProtos]++;
    freeblist(bp);
}
```

Uses BLKIPVER-261 486c, ForwDatagrams 31a, Fsrcvpcol() 99d, ICMPv6 498d, IP6HDR 498e, IPV6CLASS-260 486b, IP_VER6 232b, IPaddrlen 20c, InDelivers 31a, InDiscards 31a, InHdrErrors 31a, InReceives 31a, InUnknownProtos 31a, Link_local_scop 498e, Logip 233d, OutDiscards 31a, icmpttlexceeded6() 477a, ipforme() 125e, ipoput6() 486e, iptentative() 378d, islinklocal 498a, isv6mcast 497b, netlog() 389a, procxtns() 493a, and v6lookup() 508b.

⟨*function* ipfragfree6 492a⟩≡ (497a)

```
/*
 * ipfragfree6 - copied from ipfragfree4 - assume hold fraglock6
 */
void
ipfragfree6(IP *ip, Fragment6 *frag)
{
    Fragment6 *fl, **l;

    if(frag->blist)
        freeblist(frag->blist);

    memset(frag->src, 0, IPaddrlen);
    frag->id = 0;
    frag->blist = nil;

    l = &ip->flisthead6;
    for(fl = *l; fl; fl = fl->next) {
        if(fl == frag) {
            *l = frag->next;
            break;
        }
        l = &fl->next;
    }

    frag->next = ip->fragfree6;
    ip->fragfree6 = frag;
}
```

Uses IPaddrlen 20c.

⟨*function* ipfragallo6 492b⟩≡ (497a)

```
/*
 * ipfragallo6 - copied from ipfragalloc4
 */
Fragment6*
ipfragallo6(IP *ip)
{
    Fragment6 *f;

    while(ip->fragfree6 == nil) {
        /* free last entry on fraglist */
        for(f = ip->flisthead6; f->next; f = f->next)
            ;
        ipfragfree6(ip, f);
    }
    f = ip->fragfree6;
    ip->fragfree6 = f->next;
    f->next = ip->flisthead6;
```

```
    ip->flisthead6 = f;
    f->age = NOW + 30000;

    return f;
}
```

Uses NOW 234a and ipfragfree6() 492a.

⟨*function* procxtns 493a⟩≡ (497a)

```
static Block*
procxtns(IP *ip, Block *bp, int doreasm)
{
    int offset;
    uchar proto;
    Ip6hdr *h;

    h = (Ip6hdr *)bp->rp;
    offset = unfraglen(bp, &proto, 0);

    if(proto == FH && doreasm != 0) {
        bp = ip6reassemble(ip, offset, bp, h);
        if(bp == nil)
            return nil;
        offset = unfraglen(bp, &proto, 0);
    }

    if(proto == DOH || offset > IP6HDR)
        bp = procopts(bp);
    return bp;
}
```

Uses DOH 498d, FH 498d, IP6HDR 498e, procopts() 494a, and unfraglen() 493b.

⟨*function* unfraglen 493b⟩≡ (497a)

```
/*
 * returns length of "Unfragmentable part", i.e., sum of lengths of ipv6 hdr,
 * hop-by-hop & routing headers if present; *nexthdr is set to nexthdr value
 * of the last header in the "Unfragmentable part"; if setfh != 0, nexthdr
 * field of the last header in the "Unfragmentable part" is set to FH.
 */
int
unfraglen(Block *bp, uchar *nexthdr, int setfh)
{
    uchar *p, *q;
    int ufl, hs;

    p = bp->rp;
    q = p+6;   /* proto, = p+sizeof(Ip6hdr.vcf)+sizeof(Ip6hdr.ploadlen) */
    *nexthdr = *q;
    ufl = IP6HDR;
    p += ufl;

    while (*nexthdr == HBH || *nexthdr == RH) {
        *nexthdr = *p;
        hs = ((int)*(p+1) + 1) * 8;
        ufl += hs;
        q = p;
        p += hs;
    }

    if(*nexthdr == FH)
        *q = *p;
```

```
    if(setfh)
        *q = FH;
    return ufl;
}
```

Uses FH 498d, HBH 498d, IP6HDR 498e, and RH 498d.

⟨*function* procopts 494a⟩≡ (497a)

```
Block*
procopts(Block *bp)
{
    return bp;
}
```

⟨*function* ip6reassemble 494b⟩≡ (497a)

```
Block*
ip6reassemble(IP* ip, int uflen, Block* bp, Ip6hdr* ih)
{
    int fend, offset, ovlap, len, fragsize, pktposn;
    uint id;
    uchar src[IPaddrlen], dst[IPaddrlen];
    Block *bl, **l, *last, *prev;
    Fraghdr6 *fraghdr;
    Fragment6 *f, *fnext;

    fraghdr = (Fraghdr6 *)(bp->rp + uflen);
    memmove(src, ih->src, IPaddrlen);
    memmove(dst, ih->dst, IPaddrlen);
    id = nhgetl(fraghdr->id);
    offset = nhgets(fraghdr->offsetRM) & ~7;

    /*
     *  block lists are too hard, pullupblock into a single block
     */
    if(bp->next){
        bp = pullupblock(bp, blocklen(bp));
        ih = (Ip6hdr *)bp->rp;
    }

    qlock(&ip->fraglock6);

    /*
     *  find a reassembly queue for this fragment
     */
    for(f = ip->flisthead6; f; f = fnext){
        fnext = f->next;
        if(ipcmp(f->src, src)==0 && ipcmp(f->dst, dst)==0 && f->id == id)
            break;
        if(f->age < NOW){
            ip->stats[ReasmTimeout]++;
            ipfragfree6(ip, f);
        }
    }

    /*
     *  if this isn't a fragmented packet, accept it
     *  and get rid of any fragments that might go
     *  with it.
     */
    if(nhgets(fraghdr->offsetRM) == 0) {    /* 1st frag is also last */
        if(f) {
```

```
        ipfragfree6(ip, f);
        ip->stats[ReasmFails]++;
    }
    qunlock(&ip->fraglock6);
    return bp;
}

if(bp->base+IPFRAGSZ >= bp->rp){
    bp = padblock(bp, IPFRAGSZ);
    bp->rp += IPFRAGSZ;
}

BKFG(bp)->foff = offset;
BKFG(bp)->flen = nhgets(ih->ploadlen) + IP6HDR - uflen - IP6FHDR;

/* First fragment allocates a reassembly queue */
if(f == nil) {
    f = ipfragallo6(ip);
    f->id = id;
    memmove(f->src, src, IPaddrlen);
    memmove(f->dst, dst, IPaddrlen);

    f->blist = bp;

    qunlock(&ip->fraglock6);
    ip->stats[ReasmReqds]++;
    return nil;
}

/*
 *  find the new fragment's position in the queue
 */
prev = nil;
l = &f->blist;
bl = f->blist;
while(bl != nil && BKFG(bp)->foff > BKFG(bl)->foff) {
    prev = bl;
    l = &bl->next;
    bl = bl->next;
}

/* Check overlap of a previous fragment - trim away as necessary */
if(prev) {
    ovlap = BKFG(prev)->foff + BKFG(prev)->flen - BKFG(bp)->foff;
    if(ovlap > 0) {
        if(ovlap >= BKFG(bp)->flen) {
            freeblist(bp);
            qunlock(&ip->fraglock6);
            return nil;
        }
        BKFG(prev)->flen -= ovlap;
    }
}

/* Link onto assembly queue */
bp->next = *l;
*l = bp;

/* Check to see if succeeding segments overlap */
if(bp->next) {
```

```
    l = &bp->next;
    fend = BKFG(bp)->foff + BKFG(bp)->flen;

    /* Take completely covered segments out */
    while(*l) {
        ovlap = fend - BKFG(*l)->foff;
        if(ovlap <= 0)
            break;
        if(ovlap < BKFG(*l)->flen) {
            BKFG(*l)->flen -= ovlap;
            BKFG(*l)->foff += ovlap;
            /* move up ih hdrs */
            memmove((*l)->rp + ovlap, (*l)->rp, uflen);
            (*l)->rp += ovlap;
            break;
        }
        last = (*l)->next;
        (*l)->next = nil;
        freeblist(*l);
        *l = last;
    }
}

/*
 *  look for a complete packet.  if we get to a fragment
 *  with the trailing bit of fraghdr->offsetRM[1] set, we're done.
 */
pktposn = 0;
for(bl = f->blist; bl && BKFG(bl)->foff == pktposn; bl = bl->next) {
    fraghdr = (Fraghdr6 *)(bl->rp + uflen);
    if((fraghdr->offsetRM[1] & 1) == 0) {
        bl = f->blist;

        /* get rid of frag header in first fragment */
        memmove(bl->rp + IP6FHDR, bl->rp, uflen);
        bl->rp += IP6FHDR;
        len = nhgets(((Ip6hdr*)bl->rp)->ploadlen) - IP6FHDR;
        bl->wp = bl->rp + len + IP6HDR;
        /*
         * Pullup all the fragment headers and
         * return a complete packet
         */
        for(bl = bl->next; bl; bl = bl->next) {
            fragsize = BKFG(bl)->flen;
            len += fragsize;
            bl->rp += uflen + IP6FHDR;
            bl->wp = bl->rp + fragsize;
        }

        bl = f->blist;
        f->blist = nil;
        ipfragfree6(ip, f);
        ih = (Ip6hdr*)bl->rp;
        hnputs(ih->ploadlen, len);
        qunlock(&ip->fraglock6);
        ip->stats[ReasmOKs]++;
        return bl;
    }
    pktposn += BKFG(bl)->flen;
}
```

```
    qunlock(&ip->fraglock6);
    return nil;
}
```

⟨kernel/network/ip/ipv6.c 497a⟩≡

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "../port/error.h"

#include    "ip.h"
#include    "ipv6.h"

⟨enum _anon_ (kernel/network/ip/ipv6.c) 486a⟩

⟨macro IPV6CLASS 486b⟩
⟨macro BLKIPVER((kernel/network/ip/ipv6.c)) 486c⟩
⟨macro BKFG((kernel/network/ip/ipv6.c)) 486d⟩

Block*      ip6reassemble(IP*, int, Block*, Ip6hdr*);
Fragment6*  ipfragallo6(IP*);
void        ipfragfree6(IP*, Fragment6*);
Block*      procopts(Block *bp);
static Block*   procxtns(IP *ip, Block *bp, int doreasm);
int     unfraglen(Block *bp, uchar *nexthdr, int setfh);

⟨function ipoput6 486e⟩

⟨function ipiput6 490⟩

⟨function ipfragfree6 492a⟩

⟨function ipfragallo6 492b⟩

⟨function procxtns 493a⟩

⟨function unfraglen 493b⟩

⟨function procopts 494a⟩

⟨function ip6reassemble 494b⟩
```

kernel/network/ip/ipv6.h

⟨macro isv6mcast 497b⟩≡ (500)

```
/*
 * Internet Protocol Version 6
 *
 * rfc2460 defines the protocol, rfc2461 neighbour discovery, and
 * rfc2462 address autoconfiguration.  rfc4443 defines ICMP; was rfc2463.
 * rfc4291 defines the address architecture (including prefices), was rfc3513.
 * rfc4007 defines the scoped address architecture.
 *
 * global unicast is anything but unspecified (::), loopback (::1),
 * multicast (ff00::/8), and link-local unicast (fe80::/10).
 *
 * site-local (fec0::/10) is now deprecated, originally by rfc3879.
```

```
 *
 * Unique Local IPv6 Unicast Addresses are defined by rfc4193.
 * prefix is fc00::/7, scope is global, routing is limited to roughly a site.
 */
#define isv6mcast(addr)   ((addr)[0] == 0xff)
```

⟨*macro* islinklocal 498a⟩≡ (500)

```
#define islinklocal(addr) ((addr)[0] == 0xfe && ((addr)[1] & 0xc0) == 0x80)
```

⟨*macro* optexsts 498b⟩≡ (500)

```
#define optexsts(np)  (nhgets((np)->ploadlen) > 24)
```

⟨*macro* issmcast 498c⟩≡ (500)

```
#define issmcast(addr)  (memcmp((addr), v6solicitednode, 13) == 0)
```

⟨*enum* _anon_ (kernel/network/ip/ipv6.h) 498d⟩≡ (500)

```
enum {          /* Header Types */
  HBH   = 0,  /* hop-by-hop multicast routing protocol */
  ICMP    = 1,
  IGMP    = 2,
  GGP   = 3,
  IPINIP    = 4,
  ST    = 5,
  TCP   = 6,
  UDP   = 17,
  ISO_TP4   = 29,
  RH    = 43,
  FH    = 44,
  IDRP    = 45,
  RSVP    = 46,
  AH    = 51,
  ESP   = 52,
  ICMPv6    = 58,
  NNH   = 59,
  DOH   = 60,
  ISO_IP    = 80,
  IGRP    = 88,
  OSPF    = 89,

  Maxhdrtype  = 256,
};
```

⟨*enum* _anon_ (kernel/network/ip/ipv6.h)2 498e⟩≡ (500)

```
enum {
  /* multicast flags and scopes */

//  Well_known_flg  = 0,
//  Transient_flg = 1,

//  Interface_local_scop = 1,
  Link_local_scop = 2,
//  Site_local_scop = 5,
//  Org_local_scop  = 8,
  Global_scop = 14,

  /* various prefix lengths */
  SOLN_PREF_LEN = 13,

  /* icmpv6 unreachability codes */
  Icmp6_no_route    = 0,
```

```
  Icmp6_ad_prohib   = 1,
  Icmp6_out_src_scope = 2,
  Icmp6_adr_unreach = 3,
  Icmp6_port_unreach  = 4,
  Icmp6_gress_src_fail  = 5,
  Icmp6_rej_route   = 6,
  Icmp6_unknown   = 7,  /* our own invention for internal use */

  /* various flags & constants */
  v6MINTU   = 1280,
  HOP_LIMIT = 255,
  IP6HDR    = 40,   /* sizeof(Ip6hdr) = 8 + 2*16 */

  /* option types */

  /* neighbour discovery */
  SRC_LLADDR  = 1,
  TARGET_LLADDR = 2,
  PREFIX_INFO = 3,
  REDIR_HEADER  = 4,
  MTU_OPTION  = 5,
  /* new since rfc2461; see iana.org/assignments/icmpv6-parameters */
  V6nd_home = 8,
  V6nd_srcaddrs = 9,    /* rfc3122 */
  V6nd_ip   = 17,
  /* /lib/rfc/drafts/draft-jeong-dnsop-ipv6-dns-discovery-12.txt */
  V6nd_rdns = 25,
  /* plan 9 extensions */
  V6nd_9fs  = 250,
  V6nd_9auth  = 251,

  SRC_UNSPEC  = 0,
  SRC_UNI   = 1,
  TARG_UNI  = 2,
  TARG_MULTI  = 3,

  Tunitent  = 1,
  Tuniproxy = 2,
  Tunirany  = 3,

  /* Node constants */
  MAX_MULTICAST_SOLICIT = 3,
  RETRANS_TIMER   = 1000,
};
```

⟨*struct* Ip6hdr *(*(kernel/network/ip/ipv6.h)*)* 499a⟩≡ (500)

```
struct  Ip6hdr {
  IPV6HDR;
  uchar payload[];
};
```

⟨*struct* Fraghdr6 499b⟩≡ (500)

```
//};

struct  Fraghdr6 {
  uchar nexthdr;
  uchar res;
  uchar offsetRM[2];  /* Offset, Res, M flag */
  uchar id[4];
};
```

⟨kernel/network/ip/ipv6.h 500⟩≡

```
  ⟨macro isv6mcast 497b⟩
  ⟨macro islinklocal 498a⟩

  ⟨macro optexsts 498b⟩
  ⟨macro issmcast 498c⟩

  ⟨enum _anon_ (kernel/network/ip/ipv6.h) 498d⟩

  ⟨enum _anon_ (kernel/network/ip/ipv6.h)2 498e⟩

  typedef struct Ip6hdr Ip6hdr;
  typedef struct Opthdr Opthdr;
  typedef struct Routinghdr Routinghdr;
  typedef struct Fraghdr6 Fraghdr6;

  /* we do this in case there's padding at the end of Ip6hdr */
  #define IPV6HDR \
    uchar vcf[4];   /* version:4, traffic class:8, flow label:20 */\
    uchar ploadlen[2];  /* payload length: packet length - 40 */ \
    uchar proto;    /* next header type */ \
    uchar ttl;    /* hop limit */ \
    uchar src[IPaddrlen]; \
    uchar dst[IPaddrlen]

  ⟨struct Ip6hdr((kernel/network/ip/ipv6.h)) 499a⟩

  //struct  Opthdr {    /* unused */
  //  uchar nexthdr;
  //  uchar len;
  //};

  /*
   * Beware routing header type 0 (loose source routing); see
   * http://www.secdev.org/conf/IPv6_RH_security-csw07.pdf.
   * Type 1 is unused.  Type 2 is for MIPv6 (mobile IPv6) filtering
   * against type 0 header.
   */
  //struct  Routinghdr {    /* unused */
  //  uchar nexthdr;
  //  uchar len;
  //  uchar rtetype;
  //  uchar segrem;
  ⟨struct Fraghdr6 499b⟩

  extern uchar v6allnodesN[IPaddrlen];
  extern uchar v6allnodesL[IPaddrlen];
  //extern uchar v6allroutersN[IPaddrlen];
  //extern uchar v6allroutersL[IPaddrlen];
  extern uchar v6allnodesNmask[IPaddrlen];
  extern uchar v6allnodesLmask[IPaddrlen];
  extern uchar v6solicitednode[IPaddrlen];
  //extern uchar v6solicitednodemask[IPaddrlen];
  extern uchar v6Unspecified[IPaddrlen];
  extern uchar v6loopback[IPaddrlen];
  //extern uchar v6loopbackmask[IPaddrlen];
  extern uchar v6linklocal[IPaddrlen];
  //extern uchar v6linklocalmask[IPaddrlen];
  //extern uchar v6multicast[IPaddrlen];
  //extern uchar v6multicastmask[IPaddrlen];
```

```
extern int v6llpreflen;
//extern int v6mcpreflen;
//extern int v6snpreflen;
//extern int v6aNpreflen;
//extern int v6aLpreflen;

extern int ReTransTimer;

void ipv62smcast(uchar *, uchar *);
void icmpns(Fs *f, uchar* src, int suni, uchar* targ, int tuni, uchar* mac);
//void icmpna(Fs *f, uchar* src, uchar* dst, uchar* targ, uchar* mac, uchar flags);
void icmpttlexceeded6(Fs *f, Ipifc *ifc, Block *bp);
void icmppkttoobig6(Fs *f, Ipifc *ifc, Block *bp);
void icmphostunr(Fs *f, Ipifc *ifc, Block *bp, int code, int free);
```

Uses `Fraghdr6` 499b and `Ip6hdr` 499a.

⟨`v6tov4()` *else if ipv6 address* 501a⟩≡ (21d)

```
else {
       memset(v4, 0, 4);
       if(memcmp(v6, IPnoaddr, IPaddrlen) == 0)
           return OK_0;
       return ERROR_NEG1;
}
```

Uses `IPaddrlen` 20c and `IPnoaddr` 23b.

⟨`defmask()` *if ipv6* 501b⟩≡ (22b)

```
else {
    if(ipcmp(ip, v6loopback) == 0)
        return IPallbits;
    else if(memcmp(ip, v6linklocal, v6llpreflen) == 0)
        return v6linklocalmask;
    else if(memcmp(ip, v6solicitednode, v6snpreflen) == 0)
        return v6solicitednodemask;
    else if(memcmp(ip, v6multicast, v6mcpreflen) == 0)
        return v6multicastmask;
    return IPallbits;
}
```

Uses `IPallbits` 227d, `ipcmp` 23d, `v6linklocal-7` 504c, `v6linklocalmask-8` 504d, `v6llpreflen-9` 504e, `v6loopback-6` 504b, `v6mcpreflen-12` 505a, `v6multicast-10` 504f, `v6multicastmask-11` 504g, `v6snpreflen-15` 505d, `v6solicitednode-13` 505b, and `v6solicitednodemask-14` 505c.

⟨`Ipifc(user)` *ipv6 fields* 501c⟩≡ (24a) 501d ▷

```
Ipv6rp rp;
```

⟨`Ipifc(user)` *ipv6 fields* 501d⟩+≡ (24a) ◁ 501c

```
uchar sendra6;  /* on == send router adv */
uchar recvra6;  /* on == rcv router adv */
```

⟨`Fs(kernel)` *ipv6 fields* 501e⟩≡ (29d)

```
  v6params  *v6p;
  Route *v6root[1<<Lroot];  /* v6 routing forest */
```

⟨`IP(kernel)` *ipv6 fields* 501f⟩≡ (30c)

```
  QLock   fraglock6;
  Fragment6*  flisthead6;
  Fragment6*  fragfree6;
  Ref   id6;
```

⟨Ipifc(kernel) *ipv6 fields* 502a⟩≡ (25b)
```
  uchar sendra6;  /* flag: send router advs on this ifc */
  uchar recvra6;  /* flag: recv router advs on this ifc */
```

⟨*function* ip_init_6 502b⟩≡ (239a)
```
  void
  ip_init_6(Fs *f)
  {
      v6params *v6p;

      v6p = smalloc(sizeof(v6params));

      v6p->rp.mflag       = 0;        /* default not managed */
      v6p->rp.oflag       = 0;
      v6p->rp.maxraint    = 600000;   /* millisecs */
      v6p->rp.minraint    = 200000;
      v6p->rp.linkmtu     = 0;        /* no mtu sent */
      v6p->rp.reachtime   = 0;
      v6p->rp.rxmitra     = 0;
      v6p->rp.ttl     = MAXTTL;
      v6p->rp.routerlt    = 3 * v6p->rp.maxraint;

      v6p->hp.rxmithost   = 1000;     /* v6 RETRANS_TIMER */

      v6p->cdrouter       = -1;

      f->v6p          = v6p;
  }
```
Uses MAXTTL 81g.

⟨*struct* Ipv6rp 502c⟩≡ (222c)
```
  /* default values, one per stack */
  struct Ipv6rp
  {
      int mflag;
      int oflag;
      int maxraint;
      int minraint;
      int linkmtu;
      int reachtime;
      int rxmitra;
      int ttl;
      int routerlt;
  };
```

⟨*macro* ISIPV6MCAST 502d⟩≡ (222c)
```
  #define ISIPV6MCAST(addr) ((addr)[0] == 0xff)
```

⟨*macro* ISIPV6LINKLOCAL 502e⟩≡ (222c)
```
  #define ISIPV6LINKLOCAL(addr) ((addr)[0] == 0xfe && ((addr)[1] & 0xc0) == 0x80)
```

⟨*enum* _anon_ (include/net/ip.h) 502f⟩≡ (222c)
```
  /*
   * ipv6 constants
   * ‘ra’ is ‘router advertisement’, ‘rs’ is ‘router solicitation’.
   * ‘na’ is ‘neighbour advertisement’.
   */
  enum {
      IPV6HDR_LEN = 40,
```

```
    /* neighbour discovery option types */
    V6nd_srclladdr = 1,
    V6nd_targlladdr = 2,
    V6nd_pfxinfo = 3,
    V6nd_redirhdr = 4,
    V6nd_mtu = 5,
    /* new since rfc2461; see iana.org/assignments/icmpv6-parameters */
    V6nd_home = 8,
    V6nd_srcaddrs = 9,  /* rfc3122 */
    V6nd_ip  = 17,
    /* /lib/rfc/drafts/draft-jeong-dnsop-ipv6-dns-discovery-12.txt */
    V6nd_rdns = 25,
    /* plan 9 extensions */
    V6nd_9fs = 250,
    V6nd_9auth = 251,

    /* Router constants (all times in ms.) */
    Maxv6initraintvl= 16000,
    Maxv6initras = 3,
    Maxv6finalras = 3,
    Minv6interradelay= 3000,
    Maxv6radelay = 500,

    /* Host constants */
    Maxv6rsdelay = 1000,
    V6rsintvl = 4000,
    Maxv6rss = 3,

    /* Node constants */
    Maxv6mcastrss = 3,
    Maxv6unicastrss = 3,
    Maxv6anycastdelay= 1000,
    Maxv6na  = 3,
    V6reachabletime = 30000,
    V6retranstimer = 1000,
    V6initprobedelay= 5000,
};
```

⟨*struct* Ip6hdr 503a⟩≡ (222c)

```
/* V6 header on the wire */
struct Ip6hdr {
    uchar vcf[4];  /* version:4, traffic class:8, flow label:20 */
    uchar ploadlen[2]; /* payload length: packet length - 40 */
    uchar proto;  /* next header type */
    uchar ttl;  /* hop limit */
    ipaddr src; /* source address */
    ipaddr dst; /* destination address */
    uchar payload[];
};
```

⟨*struct* Icmp6hdr 503b⟩≡ (222c)

```
/*
 *  user-level icmpv6 with control message "headers"
 */
struct Icmp6hdr {
    uchar _0_[8];
    ipaddr laddr; /* local address */
    ipaddr raddr; /* remote address */
};
```

⟨*function* equivip6 504a⟩≡ (227a)
```
bool
equivip6(uchar *a, uchar *b)
{
    int i;

    for(i = 0; i < IPaddrlen; i++)
        if(a[i] != b[i])
            return false;
    return true;
}
```
Uses IPaddrlen 20c.

⟨*global* v6loopback 504b⟩≡ (229d)
```
static uchar v6loopback[IPaddrlen] = {
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0x01
};
```
Uses IPaddrlen 20c.

⟨*global* v6linklocal 504c⟩≡ (229d)
```
static uchar v6linklocal[IPaddrlen] = {
    0xfe, 0x80, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```
Uses IPaddrlen 20c.

⟨*global* v6linklocalmask 504d⟩≡ (229d)
```
static uchar v6linklocalmask[IPaddrlen] = {
    0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xff, 0xff,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```
Uses IPaddrlen 20c.

⟨*global* v6llpreflen 504e⟩≡ (229d)
```
static int v6llpreflen = 8; /* link-local prefix length in bytes */
```
Uses v6llpreflen-9 504e.

⟨*global* v6multicast 504f⟩≡ (229d)
```
static uchar v6multicast[IPaddrlen] = {
    0xff, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```
Uses IPaddrlen 20c.

⟨*global* v6multicastmask 504g⟩≡ (229d)
```
static uchar v6multicastmask[IPaddrlen] = {
    0xff, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0,
    0, 0, 0, 0
};
```
Uses IPaddrlen 20c.

⟨*global* v6mcpreflen 505a⟩≡ (229d)
```
  static int v6mcpreflen = 1; /* multicast prefix length */
```
Uses v6mcpreflen-12 505a.

⟨*global* v6solicitednode 505b⟩≡ (229d)
```
  static uchar v6solicitednode[IPaddrlen] = {
      0xff, 0x02, 0, 0,
      0, 0, 0, 0,
      0, 0, 0, 0x01,
      0xff, 0, 0, 0
  };
```
Uses IPaddrlen 20c.

⟨*global* v6solicitednodemask 505c⟩≡ (229d)
```
  static uchar v6solicitednodemask[IPaddrlen] = {
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff,
      0xff, 0x0, 0x0, 0x0
  };
```
Uses IPaddrlen 20c.

⟨*global* v6snpreflen 505d⟩≡ (229d)
```
  static int v6snpreflen = 13;
```
Uses v6snpreflen-15 505d.

⟨ip_init() *ipv6 init* 505e⟩≡ (42a)
```
  ip_init_6(f);
```
Uses ip_init_6() 502b.

⟨initfrag() *locals* 505f⟩≡ (42b)
```
      Fragment6 *fq6, *eq6;
```

⟨initfrag() *ipv6 init fragfree6* 505g⟩≡ (42b)
```
      ip->fragfree6 = (Fragment6*)malloc(sizeof(Fragment6) * size);
      if(ip->fragfree6 == nil)
          panic("initfrag");

      eq6 = &ip->fragfree6[size];
      for(fq6 = ip->fragfree6; fq6 < eq6; fq6++)
          fq6->next = fq6+1;

      ip->fragfree6[size-1].next = nil;
```

⟨*struct* Fragment6 505h⟩≡ (234b)
```
  struct Fragment6
  {
    Block*  blist;
    Fragment6*  next;
    uchar   src[IPaddrlen];
    uchar   dst[IPaddrlen];
    uint  id;
    ulong   age;
  };
```

⟨*struct* v6router 506a⟩≡ (234b)
```
/* one per default router known to host */
struct v6router {
  uchar inuse;
  Ipifc *ifc;
  int ifcid;
  uchar routeraddr[IPaddrlen];
  long  ltorigin;
  Routerparams  rp;
};
```

⟨*struct* v6params 506b⟩≡ (234b)
```
struct v6params
{
  Routerparams  rp;   /* v6 params, one copy per node now */
  Hostparams  hp;
  v6router  v6rlist[3]; /* max 3 default routers, currently */
  int   cdrouter; /* uses only v6rlist[cdrouter] if   */
          /* cdrouter >= 0. */
};
```

⟨*struct* V6route 506c⟩≡ (234b)
```
struct V6route
{
  ulong address[IPllen];
  ulong endaddress[IPllen];
  uchar gate[IPaddrlen];
};
```

⟨*function* etherread6 506d⟩≡ (366c)
```
/*
 *  process to read from the ethernet, IPv6
 */
static void
etherread6(void *a)
{
    Ipifc *ifc;
    Block *bp;
    Etherrock *er;

    ifc = a;
    er = ifc->arg;
    er->read6p = up;    /* hide identity under a rock for unbind */
    if(waserror()){
        er->read6p = 0;
        pexit("hangup", 1);
    }
    for(;;){
        bp = devtab[er->mchan6->type]->bread(er->mchan6, ifc->maxtu, 0);
        if(!canrlock(ifc)){
            freeb(bp);
            continue;
        }
        if(waserror()){
            runlock(ifc);
            nexterror();
        }
        ifc->in++;
        bp->rp += ifc->m->hsize;
        if(ifc->lifc == nil)
```

```
            freeb(bp);
        else
            ipiput6(er->f, ifc, bp);
        runlock(ifc);
        poperror();
    }
}
```

Uses ipiput6() 490.

⟨*global* v6freelist 507a⟩≡ (369c)

```
static Route*   v6freelist;
```

⟨*macro* V6H 507b⟩≡ (369c)

```
#define V6H(a)  (((a)[IPllen-1] & 0x07ffffff)>>(32-Lroot-5))
```

⟨*function* v6addroute 507c⟩≡ (369c)

```
//#define ISDFLT(a, mask, tag) ((ipcmp((a),v6Unspecified)==0) && (ipcmp((mask),v6Unspecified)==0) && (strcmp((t

void
v6addroute(Fs *f, char *tag, uchar *a, uchar *mask, uchar *gate, int type)
{
    Route *p;
    ulong sa[IPllen], ea[IPllen];
    ulong x, y;
    int h, eh;

    /*
    if(ISDFLT(a, mask, tag))
        f->v6p->cdrouter = -1;
    */


    for(h = 0; h < IPllen; h++){
        x = nhgetl(a+4*h);
        y = nhgetl(mask+4*h);
        sa[h] = x & y;
        ea[h] = x | ~y;
    }

    eh = V6H(ea);
    for(h = V6H(sa); h <= eh; h++) {
        p = allocroute(type);
        memmove(p->v6.address, sa, IPaddrlen);
        memmove(p->v6.endaddress, ea, IPaddrlen);
        memmove(p->v6.gate, gate, IPaddrlen);
        memmove(p->tag, tag, sizeof(p->tag));

        wlock(&routelock);
        addnode(f, &f->v6root[h], p);
        while(p = f->queue) {
            f->queue = p->mid;
            walkadd(f, &f->v6root[h], p->left);
            freeroute(p);
        }
        wunlock(&routelock);
    }
    v6routegeneration++;

    ipifcaddroute(f, 0, a, mask, gate, type);
}
```

Uses IPaddrlen 20c, IPllen 232b, V6H-132 507b, addnode() 117, allocroute() 115c, freeroute() 116a, ipifcaddroute() 133b, routelock-123 115a, v6routegeneration-125 519b, and walkadd() 118d.

⟨*function* v6delroute 508a⟩≡ (369c)

```
void
v6delroute(Fs *f, uchar *a, uchar *mask, int dolock)
{
    Route **r, *p;
    Route rt;
    int h, eh;
    ulong x, y;

    for(h = 0; h < IPllen; h++){
        x = nhgetl(a+4*h);
        y = nhgetl(mask+4*h);
        rt.v6.address[h] = x & y;
        rt.v6.endaddress[h] = x | ~y;
    }
    rt.type = 0;

    eh = V6H(rt.v6.endaddress);
    for(h=V6H(rt.v6.address); h<=eh; h++) {
        if(dolock)
            wlock(&routelock);
        r = looknode(&f->v6root[h], &rt);
        if(r) {
            p = *r;
            if(--(p->ref) == 0){
                *r = 0;
                addqueue(&f->queue, p->left);
                addqueue(&f->queue, p->mid);
                addqueue(&f->queue, p->right);
                freeroute(p);
                while(p = f->queue) {
                    f->queue = p->mid;
                    walkadd(f, &f->v6root[h], p->left);
                    freeroute(p);
                }
            }
        }
        if(dolock)
            wunlock(&routelock);
    }
    v6routegeneration++;

    ipifcremroute(f, 0, a, mask);
}
```

Uses IPllen 232b, V6H-132 507b, addqueue() 118e, freeroute() 116a, ipifcremroute() 133c, looknode() 131b, routelock-123 115a, v6routegeneration-125 519b, and walkadd() 118d.

⟨*function* v6lookup 508b⟩≡ (369c)

```
Route*
v6lookup(Fs *f, uchar *a, Conv *c)
{
    Route *p, *q;
    ulong la[IPllen];
    int h;
    ulong x, y;
    uchar gate[IPaddrlen];
    Ipifc *ifc;
```

```
	if(memcmp(a, v4prefix, IPv4off) == 0){
		q = v4lookup(f, a+IPv4off, c);
		if(q != nil)
			return q;
	}

	if(c != nil && c->r != nil && c->r->ifc != nil && c->rgen == v6routegeneration)
		return c->r;

	for(h = 0; h < IPllen; h++)
		la[h] = nhgetl(a+4*h);

	q = 0;
	for(p=f->v6root[V6H(la)]; p;){
		for(h = 0; h < IPllen; h++){
			x = la[h];
			y = p->v6.address[h];
			if(x == y)
				continue;
			if(x < y){
				p = p->left;
				goto next;
			}
			break;
		}
		for(h = 0; h < IPllen; h++){
			x = la[h];
			y = p->v6.endaddress[h];
			if(x == y)
				continue;
			if(x > y){
				p = p->right;
				goto next;
			}
			break;
		}
		q = p;
		p = p->mid;
next:		;
	}

	if(q && (q->ifc == nil || q->ifcid != q->ifc->ifcid)){
		if(q->type & Rifc) {
			for(h = 0; h < IPllen; h++)
				hnputl(gate+4*h, q->v6.address[h]);
			ifc = findipifc(f, gate, q->type);
		} else
			ifc = findipifc(f, q->v6.gate, q->type);
		if(ifc == nil)
			return nil;
		q->ifc = ifc;
		q->ifcid = ifc->ifcid;
	}
	if(c != nil){
		c->r = q;
		c->rgen = v6routegeneration;
	}

	return q;
```

```
}
```

Uses `IPaddrlen` 20c, `IPllen` 232b, `IPv4off` 21a, `Rifc` 38e, `V6H-132` 507b, `findipifc()` 113e, `v4lookup()` 112a, `v4prefix` 20e, and `v6routegeneration-125` 519b.

⟨*function* `ipifcra6` 510a⟩≡ (384b)

```
char*
ipifcra6(Ipifc *ifc, char **argv, int argc)
{
    int i, argsleft, vmax = ifc->rp.maxraint, vmin = ifc->rp.minraint;

    argsleft = argc - 1;
    i = 1;

    if(argsleft % 2 != 0)
        return Ebadarg;

    while (argsleft > 1) {
        if(strcmp(argv[i], "recvra") == 0)
            ifc->recvra6 = (atoi(argv[i+1]) != 0);
        else if(strcmp(argv[i], "sendra") == 0)
            ifc->sendra6 = (atoi(argv[i+1]) != 0);
        else if(strcmp(argv[i], "mflag") == 0)
            ifc->rp.mflag = (atoi(argv[i+1]) != 0);
        else if(strcmp(argv[i], "oflag") == 0)
            ifc->rp.oflag = (atoi(argv[i+1]) != 0);
        else if(strcmp(argv[i], "maxraint") == 0)
            ifc->rp.maxraint = atoi(argv[i+1]);
        else if(strcmp(argv[i], "minraint") == 0)
            ifc->rp.minraint = atoi(argv[i+1]);
        else if(strcmp(argv[i], "linkmtu") == 0)
            ifc->rp.linkmtu = atoi(argv[i+1]);
        else if(strcmp(argv[i], "reachtime") == 0)
            ifc->rp.reachtime = atoi(argv[i+1]);
        else if(strcmp(argv[i], "rxmitra") == 0)
            ifc->rp.rxmitra = atoi(argv[i+1]);
        else if(strcmp(argv[i], "ttl") == 0)
            ifc->rp.ttl = atoi(argv[i+1]);
        else if(strcmp(argv[i], "routerlt") == 0)
            ifc->rp.routerlt = atoi(argv[i+1]);
        else
            return Ebadarg;

        argsleft -= 2;
        i += 2;
    }

    /* consistency check */
    if(ifc->rp.maxraint < ifc->rp.minraint) {
        ifc->rp.maxraint = vmax;
        ifc->rp.minraint = vmin;
        return Ebadarg;
    }
    return nil;
}
```

⟨*enum* `_anon_ (kernel/network/ip/ipifc.c)3` 510b⟩≡ (384b)

```
enum {
    unknownv6,      /* UGH */
//  multicastv6,
    unspecifiedv6,
```

```
    linklocalv6,
    globalv6,
};
```

⟨*function* v6addrtype 511a⟩≡ (384b)

```
int
v6addrtype(uchar *addr)
{
    if(isv4(addr) || ipcmp(addr, IPnoaddr) == 0)
        return unknownv6;
    else if(islinklocal(addr) ||
        isv6mcast(addr) && (addr[1] & 0xF) <= Link_local_scop)
        return linklocalv6;
    else
        return globalv6;
}
```

Uses IPnoaddr 23b, Link_local_scop 498e, globalv6-112 510b, ipcmp 23d, islinklocal 498a, isv4() 21b, isv6mcast 497b, linklocalv6-111 510b, and unknownv6-109 510b.

⟨*macro* v6addrcurr 511b⟩≡ (384b)

```
#define v6addrcurr(lifc) ((lifc)->preflt == ~0L || \
            (lifc)->origint + (lifc)->preflt >= NOW/1000)
```

⟨*function* findprimaryipv6 511c⟩≡ (384b)

```
static void
findprimaryipv6(Fs *f, uchar *local)
{
    int atype, atypel;
    Conv **cp, **e;
    Ipifc *ifc;
    Iplifc *lifc;

    ipmove(local, v6Unspecified);
    atype = unspecifiedv6;

    /*
     * find "best" (global > link local > unspecified)
     * local address; address must be current.
     */
    e = &f->ipifc->conv[f->ipifc->nc];
    for(cp = f->ipifc->conv; cp < e; cp++){
        if(*cp == 0)
            continue;
        ifc = (Ipifc*)(*cp)->ptcl;
        for(lifc = ifc->lifc; lifc; lifc = lifc->next){
            atypel = v6addrtype(lifc->local);
            if(atypel > atype && v6addrcurr(lifc)) {
                ipmove(local, lifc->local);
                atype = atypel;
                if(atype == globalv6)
                    return;
            }
        }
    }
}
```

Uses globalv6-112 510b, ipmove 23c, unspecifiedv6-110 510b, v6Unspecified 371a, v6addrcurr-113 511b, and v6addrtype() 511a.

⟨*function* ipv6local 512a⟩≡ (384b)
```
/*
 *  return first v6 address associated with an interface
 */
int
ipv6local(Ipifc *ifc, uchar *addr)
{
    Iplifc *lifc;

    for(lifc = ifc->lifc; lifc; lifc = lifc->next){
        if(!isv4(lifc->local) && !(lifc->tentative)){
            ipmove(addr, lifc->local);
            return 1;
        }
    }
    return 0;
}
```
Uses ipmove 23c and isv4() 21b.

⟨*function* ipv6anylocal 512b⟩≡ (384b)
```
int
ipv6anylocal(Ipifc *ifc, uchar *addr)
{
    Iplifc *lifc;

    for(lifc = ifc->lifc; lifc; lifc = lifc->next){
        if(!isv4(lifc->local)){
            ipmove(addr, lifc->local);
            return SRC_UNI;
        }
    }
    return SRC_UNSPEC;
}
```
Uses SRC_UNI 498e, SRC_UNSPEC 498e, ipmove 23c, and isv4() 21b.

⟨*function* ipifcadd6 512c⟩≡ (384b)
```
char*
ipifcadd6(Ipifc *ifc, char**argv, int argc)
{
    int plen = 64;
    long origint = NOW / 1000, preflt = ~0L, validlt = ~0L;
    char addr[40], preflen[6];
    char *params[3];
    uchar autoflag = 1, onlink = 1;
    uchar prefix[IPaddrlen];
    Iplifc *lifc;

    switch(argc) {
    case 7:
        preflt = atoi(argv[6]);
        /* fall through */
    case 6:
        validlt = atoi(argv[5]);
        /* fall through */
    case 5:
        autoflag = atoi(argv[4]);
        /* fall through */
    case 4:
        onlink = atoi(argv[3]);
        /* fall through */
```

```
    case 3:
        plen = atoi(argv[2]);
        /* fall through */
    case 2:
        break;
    default:
        return Ebadarg;
    }

    if (parseip(prefix, argv[1]) != 6 || validlt < preflt || plen < 0 ||
        plen > 64 || islinklocal(prefix))
        return Ebadarg;

    lifc = smalloc(sizeof(Iplifc));
    lifc->onlink = (onlink != 0);
    lifc->autoflag = (autoflag != 0);
    lifc->validlt = validlt;
    lifc->preflt = preflt;
    lifc->origint = origint;

    /* issue "add" ctl msg for v6 link-local addr and prefix len */
    if(!ifc->m->pref2addr)
        return Ebadarg;
    ifc->m->pref2addr(prefix, ifc->mac);    /* mac → v6 link-local addr */
    snprint(addr, sizeof addr, "%I", prefix);
    snprint(preflen, sizeof preflen, "/%d", plen);
    params[0] = "add";
    params[1] = addr;
    params[2] = preflen;

    return ipifcadd(ifc, params, 3, 0, lifc);
}
```

Uses IPaddrlen 20c, NOW 234a, ipifcadd() 71c, islinklocal 498a, and parseip() 89.

⟨Medium(kernel) *ipv6 methods* 513a⟩≡ (26c)

```
/* v6 address generation */
void  (*pref2addr)(uchar *pref, uchar *ea);
```

⟨*struct* Udp6hdr 513b⟩≡ (405)

```
struct Udp6hdr {
    uchar viclfl[4];
    uchar len[2];
    uchar nextheader;
    uchar hoplimit;
    uchar udpsrc[IPaddrlen];
    uchar udpdst[IPaddrlen];

    /* udp header */
    uchar   udpsport[2];    /* Source port */
    uchar   udpdport[2];    /* Destination port */
    uchar   udplen[2];  /* data length */
    uchar   udpcksum[2];    /* Checksum */
};
```

Uses IPaddrlen 20c.

⟨udpkick() *locals* 513c⟩+≡ (140b) ◁148g

```
Udp6hdr *uh6;
```

⟨udpkick() *switch version ipv6 case* 514a⟩≡ (140b)

```
  case V6:
      bp = padblock(bp, UDP6_IPHDR_SZ+UDP_UDPHDR_SZ);
      if(bp == nil)
          return;

      /*
       * using the v6 ip header to create pseudo header
       * first then reset it to the normal ip header
       */
      uh6 = (Udp6hdr *)(bp->rp);
      memset(uh6, 0, 8);
      ptcllen = dlen + UDP_UDPHDR_SZ;
      hnputl(uh6->viclfl, ptcllen);
      uh6->hoplimit = IP_UDPPROTO;
      if(ucb->headers) {
          ipmove(uh6->udpdst, raddr);
          hnputs(uh6->udpdport, rport);
          ipmove(uh6->udpsrc, laddr);
          rc = nil;
      } else {
          ipmove(uh6->udpdst, c->raddr);
          hnputs(uh6->udpdport, c->rport);
          if(ipcmp(c->laddr, IPnoaddr) == 0)
              findlocalip(f, c->laddr, c->raddr);
          ipmove(uh6->udpsrc, c->laddr);
          rc = c;
      }
      hnputs(uh6->udpsport, c->lport);
      hnputs(uh6->udplen, ptcllen);
      uh6->udpcksum[0] = 0;
      uh6->udpcksum[1] = 0;
      hnputs(uh6->udpcksum,
             ptclcsum(bp, UDP6_PHDR_OFF, dlen+UDP_UDPHDR_SZ+UDP6_PHDR_SZ));
      memset(uh6, 0, 8);
      uh6->viclfl[0] = IP_VER6;
      hnputs(uh6->len, ptcllen);
      uh6->nextheader = IP_UDPPROTO;
      ipoput6(f, bp, 0, c->ttl, c->tos, rc);
      break;
```

Uses IP_UDPPROTO-371 135a, IP_VER6 232b, IPnoaddr 23b, UDP6_IPHDR_SZ-368 514c, UDP6_PHDR_OFF-370 514c, UDP6_PHDR_SZ-369 514c, UDP_UDPHDR_SZ-364 138c, V6 21g, findlocalip() 379, ipcmp 23d, ipmove 23c, ipoput6() 486e, and ptclcsum() 90d.

⟨udpkick() *set version to V4 or V6* 514b⟩≡ (140b)

```
  ⟨udpkick() set version to V4 or V6 if special headers 516a⟩
  else {
      if( (memcmp(c->raddr, v4prefix, IPv4off) == 0 &&
          memcmp(c->laddr, v4prefix, IPv4off) == 0)
          || ipcmp(c->raddr, IPnoaddr) == 0)
          version = V4;
      else
          version = V6;
  }
```

Uses IPnoaddr 23b, IPv4off 21a, V4 21g, V6 21g, ipcmp 23d, and v4prefix 20e.

⟨*constant* UDP6_xxx 514c⟩≡ (403b)

```
  UDP6_IPHDR_SZ = 40,
  UDP6_PHDR_SZ = 40,
```

```
  UDP6_PHDR_OFF = 0,
```

⟨ipiput4() *call ipiput6 if block is not ipv4* 515a⟩≡ (98)

```
  if(BLKIPVER(bp) != IP_VER4) {
      ipiput6(f, ifc, bp);
      return;
  }
```

Uses BLKIPVER-264 39c, IP_VER4 94a, and ipiput6() 490.

⟨udpiput() *locals* 515b⟩+≡ (142) ◁145a

```
  Udp6hdr *uh6;
  int oviclfl;
```

⟨udpiput() *checking checksum ipv6 case* 515c⟩≡ (145b)

```
  case V6:
      uh6 = (Udp6hdr*)(bp->rp);
      len = nhgets(uh6->udplen);
      oviclfl = nhgetl(uh6->viclfl);
      olen = nhgets(uh6->len);
      ottl = uh6->hoplimit;
      ipmove(raddr, uh6->udpsrc);
      ipmove(laddr, uh6->udpdst);
      lport = nhgets(uh6->udpdport);
      rport = nhgets(uh6->udpsport);
      memset(uh6, 0, 8);
      hnputl(uh6->viclfl, len);
      uh6->hoplimit = IP_UDPPROTO;
      if(ptclcsum(bp, UDP6_PHDR_OFF, len+UDP6_PHDR_SZ)) {
          upriv->ustats.udpInErrors++;
          netlog(f, Logudp, "udp: checksum error %I\n", raddr);
          DPRINT("udp: checksum error %I\n", raddr);
          freeblist(bp);
          return;
      }
      hnputl(uh6->viclfl, oviclfl);
      hnputs(uh6->len, olen);
      uh6->nextheader = IP_UDPPROTO;
      uh6->hoplimit = ottl;
      break;
```

Uses DPRINT-363 403a, IP_UDPPROTO-371 135a, Logudp 233d, UDP6_PHDR_OFF-370 514c, UDP6_PHDR_SZ-369 514c, V6 21g, ipmove 23c, netlog() 389a, and ptclcsum() 90d.

⟨udpiput() *no conversation found, ipv6 case* 515d⟩≡ (146b)

```
  case V6:
      icmphostunr(f, ifc, bp, Icmp6_port_unreach, 0);
      break;
```

Uses Icmp6_port_unreach 498e, V6 21g, and icmphostunr() 476.

⟨udpiput() *trim the packet, ipv6 case* 515e⟩≡ (146a)

```
  case V6:
      bp = trimblock(bp, UDP6_IPHDR_SZ+UDP_UDPHDR_SZ, len);
      break;
```

Uses UDP6_IPHDR_SZ-368 514c, UDP_UDPHDR_SZ-364 138c, and V6 21g.

⟨rr() *else if ipv6 address* 515f⟩≡ (123a)

```
  else
      h = V6H(r->v6.address);
```

Uses V6H-132 507b.

⟨udpkick() *set version to V4 or V6 if special headers* 516a⟩≡ (514b)
```
  if(ucb->headers) {
      if(memcmp(laddr, v4prefix, IPv4off) == 0
      || ipcmp(laddr, IPnoaddr) == 0)
          version = V4;
      else
          version = V6;
  }
```
Uses IPnoaddr 23b, IPv4off 21a, V4 21g, V6 21g, ipcmp 23d, and v4prefix 20e.

⟨udpiput() *new conv to create, ipv6 case* 516b⟩≡ (148b)
```
  case V6:
      ipmove(laddr, ifc->lifc->local);
      break;
```
Uses V6 21g and ipmove 23c.

⟨udpiput() *set version to V4 or V6* 516c⟩≡ (142)
```
  version = ((uh4->vihl&0xF0)==IP_VER6) ? V6 : V4;
```
Uses IP_VER6 232b, V4 21g, and V6 21g.

⟨Route *ipv6 route union case* 516d⟩≡ (37e)
```
  V6route v6;
```

⟨Conv(kernel) *ipv6 fields* 516e⟩≡ (33e)
```
  // enum<v6_or_v4>
  uchar ipversion;
```

⟨Fsstdconnect() *set ipversion field to V4 or V6* 516f⟩≡ (56b)
```
  if( (memcmp(c->raddr, v4prefix, IPv4off) == 0 &&
       memcmp(c->laddr, v4prefix, IPv4off) == 0)
      || ipcmp(c->raddr, IPnoaddr) == 0)
      c->ipversion = V4;
  else
      c->ipversion = V6;
```
Uses IPnoaddr 23b, IPv4off 21a, V4 21g, V6 21g, ipcmp 23d, and v4prefix 20e.

⟨ipifcadd() *add route if ipv6 case* 516g⟩≡ (71c)
```
  else
      v6addroute(f, tifc, rem, mask, rem, type);
```
Uses tifc-104 73a and v6addroute() 507c.

⟨Iplifc(user) *ipv6 fields* 516h⟩≡ (24e)
```
  ulong preflt;   /* preferred lifetime */
  ulong validlt;  /* valid lifetime */
```

⟨Iplifc(kernel) *ipv6 fields* 516i⟩≡ (25e) 516j ▷
```
  long  preflt;   /* v6 preferred lifetime */
  long  validlt;  /* v6 valid lifetime */
```

⟨Iplifc(kernel) *ipv6 fields* 516j⟩+≡ (25e) ◁ 516i
```
  uchar onlink;   /* =1 => onlink, =0 offlink. */
  uchar tentative;  /* =1 => v6 dup disc on, =0 => confirmed unique */
  uchar autoflag; /* v6 autonomous flag */
  long  origint;  /* time when addr was added */
```

⟨ipifcadd() *set tentative for ipv6* 516k⟩≡ (71c)
```
      if(isv4(ip))
          tentative = false;
```
Uses isv4() 21b.

⟨ipifcctl() *else if other string* 517a⟩+≡ (71a) ◁180f 517b▷
```
  else if(strcmp(argv[0], "try") == 0)
      return ipifcadd(ifc, argv, argc, true, nil);
```
Uses ipifcadd() 71c.

⟨ipifcctl() *else if other string* 517b⟩+≡ (71a) ◁517a 517c▷
```
  else if(strcmp(argv[0], "add6") == 0)
      return ipifcadd6(ifc, argv, argc);
```
Uses ipifcadd6() 512c.

⟨ipifcctl() *else if other string* 517c⟩+≡ (71a) ◁517b
```
  else if(strcmp(argv[0], "ra6") == 0)
      return ipifcra6(ifc, argv, argc);
```
Uses ipifcra6() 510a.

⟨ipifcadd() *if ipv6 tentative and broacast* 517d⟩≡ (71c)
```
  if(tentative && sendnbrdisc)
      icmpns(f, 0, SRC_UNSPEC, ip, TARG_MULTI, ifc->mac);
```
Uses SRC_UNSPEC 498e, TARG_MULTI 498e, and icmpns() 474b.

⟨ipifcadd() *locals* 517e⟩+≡ (71c) ◁178c
```
  bool sendnbrdisc = false;
```

⟨ipifcadd() *if ipv6 add multicast addresses to self cache* 517f⟩≡ (71c)
```
  else {
      if(ipcmp(ip, v6loopback) == 0) {
          /* add node-local mcast address */
          addselfcache(f, ifc, lifc, v6allnodesN, Rmulti);

          /* add route for all node multicast */
          v6addroute(f, tifc, v6allnodesN, v6allnodesNmask,
              v6allnodesN, Rmulti);
      }

      /* add all nodes multicast address */
      addselfcache(f, ifc, lifc, v6allnodesL, Rmulti);

      /* add route for all nodes multicast */
      v6addroute(f, tifc, v6allnodesL, v6allnodesLmask, v6allnodesL,
          Rmulti);

      /* add solicited-node multicast address */
      ipv62smcast(bcast, ip);
      addselfcache(f, ifc, lifc, bcast, Rmulti);

      sendnbrdisc = true;
  }
```
Uses Rmulti 38e, addselfcache() 127b, ipcmp 23d, ipv62smcast() 372e, tifc-104 73a, v6addroute() 507c, v6allnodesL 372a, v6allnodesLmask 372b, v6allnodesN 371e, v6allnodesNmask 371f, and v6loopback 371b.

⟨ipifcadd() *set ipv6 fields for lifc* 517g⟩≡ (71c)
```
  lifc->tentative = tentative;
  if(lifcp) {
      lifc->onlink = lifcp->onlink;
      lifc->autoflag = lifcp->autoflag;

      lifc->validlt = lifcp->validlt;
      lifc->preflt = lifcp->preflt;
      lifc->origint = lifcp->origint;
```

```
  } else {        /* default values */
      lifc->onlink = lifc->autoflag = 1;
      lifc->validlt = lifc->preflt = ~0L;
      lifc->origint = NOW / 1000;
  }
```

Uses NOW 234a.

⟨ipifcadd() *when already local address for ifc, copy ipv6 fields* 518a⟩≡ (73b)

```
  if(lifc->tentative != tentative)
      lifc->tentative = tentative;
  if(lifcp) {
      lifc->onlink = lifcp->onlink;
      lifc->autoflag = lifcp->autoflag;

      lifc->validlt = lifcp->validlt;
      lifc->preflt = lifcp->preflt;
      lifc->origint = lifcp->origint;
  }
```

⟨ipifcremlifc() *if ipv6 local* 518b⟩≡ (82c)

```
  else {
      v6delroute(f, lifc->remote, lifc->mask, 1);
      if(ipcmp(lifc->local, v6loopback) == 0)
          /* remove route for all node multicast */
          v6delroute(f, v6allnodesN, v6allnodesNmask, 1);
      else if(memcmp(lifc->local, v6linklocal, v6llpreflen) == 0)
          /* remove route for all link multicast */
          v6delroute(f, v6allnodesL, v6allnodesLmask, 1);
  }
```

⟨allocroute() *if ipv6 route* 518c⟩≡ (115c)

```
  else {
      n = sizeof(RouteTree) + sizeof(V6route);
      l = &v6freelist;
  }
```

Uses v6freelist-124 507a.

⟨freeroute() *if ipv6 route* 518d⟩≡ (116a)

```
  else
      l = &v6freelist;
```

Uses v6freelist-124 507a.

⟨rangecompare() *if ipv6 routes* 518e⟩≡ (116c)

```
  if(lcmp(a->v6.endaddress, b->v6.address) < 0)
      return Rpreceeds;

  if(lcmp(a->v6.address, b->v6.endaddress) > 0)
      return Rfollows;

  if(lcmp(a->v6.address, b->v6.address) <= 0
  && lcmp(a->v6.endaddress, b->v6.endaddress) >= 0){
      if(lcmp(a->v6.address, b->v6.address) == 0
      && lcmp(a->v6.endaddress, b->v6.endaddress) == 0)
              return Requals;
      return Rcontains;
  }

  return Rcontained;
```

Uses Rcontained-130 116b, Rcontains-129 116b, Requals-128 116b, Rfollows-127 116b, Rpreceeds-126 116b, and lcmp() 519a.

⟨*function* `lcmp` 519a⟩≡ (369c)

```
/*
 *   compare 2 v6 addresses
 */
static int
lcmp(ulong *a, ulong *b)
{
    int i;

    for(i = 0; i < IPllen; i++){
        if(a[i] > b[i])
            return 1;
        if(a[i] < b[i])
            return -1;
    }
    return 0;
}
```

Uses `IPllen` 232b.

⟨*global* `v6routegeneration` 519b⟩≡ (369c)

```
static ulong v6routegeneration;
```

Glossary

MTU
TTL
TOS
IP
UDP
TCP
IL
IFC
ICMP

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Ab-463: 201c, 211c, 268c
Abt-456: 205b, 268b
ACK-287: 406, 428, 429, 430, 434b, 441, 449, 453
AckDelay-61: 157, 391b, 397b, 398a
ACTIVE-311: 406, 421
activemulti(): 252c, 257b, 319d
ActiveOpens-333: 410c, 421
addethercard(): 261a
addipmedium(): 27c, 216e, 217c, 366a, 369a
addnode(): 114, 117, 117, 118d, 507c
addqueue(): 117, 118e, 131a, 508a
addreseq(): 441, 458c
Addrlen: 232b
AddrMaskReply-166: 469c
AddrMaskReply-200: 343a
AddrMaskRequest-165: 469c
AddrMaskRequest-199: 343a
AddrmxpErrs6-144: 468, 478
addselfcache(): 71c, 127b, 179a, 382a, 383c, 517f
aendian-17: 230b, 230b
aendian-186: 390b, 390b
aformat: 108d, 340a, 340a
AGain-68: 391b, 392e
Agnt-606: 273c, 300
AH: 498d
Ait-514: 271b, 286
Alinelen-120: 108d, 339
allmar-475: 211a, 211c
allocroute(): 114, 115c, 118e, 507c
Am-464: 201c, 211c, 268c
Ana10FD: 311d, 313, 315c
Ana10HD: 311d, 313, 315c
AnaAck: 315c
AnaAP: 296b, 311d, 313, 315c
AnaNp: 315c
AnaP: 296b, 311d, 313, 315c
Anar: 311d, 314b
AnaRf: 315c

AnaT4: 311d, 315c
AnaTXFD: 311d, 313, 315c
AnaTXHD: 311d, 313, 315c
Anc-676: 275c
Aner: 314b
Anlpar: 313, 314b
announcectlmsg(): 57b, 57c
Announced: 34b, 58a, 62g, 142, 329d, 419d
announced(): 57c, 58a
Announcing: 34b, 57c
Annprr: 314b
Annptr: 314b
AOK-117: 333a, 337b, 338
Ar-462: 268c
Areq-605: 273c, 300
Arm-438: 267c
Arp: 106b, 234b
Arp.cache: 106b
Arp.dropf: 106b
Arp.dropl: 106b
Arp.f: 106c
Arp.hash: 106b
Arp.rxmitp: 106b
Arp.rxmt: 106b
Arp.rxmtq: 106b
Arp (typedef): 234b
Arpen-735: 277a, 301
Arpent: 234b, 234b
Arpent (typedef): 234b
arpenter(): 27f, 338, 356c, 363, 383c, 481b
arpget(): 336, 358
arpinit(): 40b, 107b
arpread(): 108c, 108d
arprelease(): 337a, 360c, 361
ARPREPLY-183: 357b, 363
ARPREQUEST-182: 357b, 360c, 362, 363
arpresolve(): 337b, 365b
arpstate: 108d, 333b
arpwrite(): 109a, 109b
Asdchk-616: 274a
Asde-559: 272, 285b
Asdv10-598: 273b
Asdv100-599: 273b
Asdv1000-600: 273b
AsdvMASK-596: 273b
AsdvSHIFT-597: 273b
at93c46io(): 298, 300
at93c46r(): 300, 302b
Atd-452: 268a

attach(): 199b, 201c
ATTACHER-250: 41b, 49c, 52a, 62g, 193b, 328c
AWAIT-118: 333a, 336, 338
backoff(): 419c, 463a
balancetree(): 117, 119b
Bam-697: 275d, 289b
Bem-556: 272
bindctlmsg(): 61a, 61b
BKFG-262: 486d
BKFG-267: 100c, 238
BLKIP-266: 39d, 100c
BLKIPVER-261: 486c, 490
BLKIPVER-264: 39c, 515a
Bmcr: 311c, 311d, 314b
BmcrAne: 311d, 315a
BmcrCte: 315a
BmcrDm: 315a
BmcrI: 315a
BmcrLe: 315a
BmcrPd: 315a
BmcrR: 311c, 311d, 315a
BmcrRan: 311d, 315a
BmcrSs0: 315a
BmcrSs1: 315a
Bmsr: 310, 311d, 313, 314b
Bmsr100T2FD: 315b
Bmsr100T2HD: 315b
Bmsr100T4: 315b
Bmsr100TXFD: 311d, 315b
Bmsr100TXHD: 311d, 315b
Bmsr10TFD: 311d, 315b
Bmsr10THD: 311d, 315b
BmsrAna: 311d, 313, 315b
BmsrAnc: 313, 315b
BmsrEc: 315b
BmsrEs: 311d, 315b
BmsrJd: 315b
BmsrLs: 313, 315b
BmsrPs: 315b
BmsrRf: 315b
Bnry-384: 200a, 207, 266b
Bos-435: 267c
Bsex-709: 275d
Bsize1024-700: 275d
Bsize16384-703: 275d
Bsize2048-699: 275d, 289b
Bsize256-702: 275d
Bsize512-701: 275d
BsizeMASK-698: 275d

Bus64-588: 273a
calcd(): 119b, 119c
call(): 353c, 353d
cards-796: 260b, 261a, 261c, 263
cards-799: 323a, 323b, 323d
Cdh-459: 205b, 268b
Ce-746: 277e
Cf-621: 274b
Cfi-706: 275d
Cfien-705: 275d
chandial(): 77c, 353c
classmask-5: 22b, 22c
Clda0-382: 266b
Clda1-383: 266b
cleanarpent(): 335, 336, 340c
CLONE-309: 406, 421, 434b
closeconv(): 50b, 50c, 62g, 63b
Closed-315: 406, 412c, 413a, 414a, 414b, 414c, 416a, 419d, 441, 449, 454b, 456, 461
Close_wait-322: 406, 412c, 414c, 415a, 441
Closing-323: 406, 441
CMclear-256: 191b, 191c, 191d
CMonly-257: 191b, 191c, 191d
CMrdtr-793: 282a, 282b, 282c
CMset-255: 191b, 191c, 191d
Cnt-431: 201c, 205b, 267b
Col-455: 268b
ColdMASK-716: 276a, 285b
ColdSHIFT-717: 276a, 285b, 286
connectctlmsg(): 55b, 55c
Connected: 34b, 56a, 144, 329d
connected(): 55c, 56a
Connecting: 34b, 55c
Conv: 234b, 234b
CONV-246: 35d, 48a, 49c, 50b, 54d, 55a, 61d, 61e, 62d, 62g, 64g, 65c, 65e, 193b, 193c, 193d, 328c, 329b, 330b
Conv (typedef): 234b
conversation_state: 34b
convmac(): 108d, 340b
convroute(): 121b, 121c, 125b, 132b
copygate(): 117, 118a
Cr-381: 199b, 200a, 200b, 201a, 203, 204, 207, 208a, 208b, 209, 211c, 266b
Crc-445: 268a
Crce-468: 207, 268d
Crda0-389: 204, 266b
Crda1-390: 204, 266b
Crs-457: 205b, 268b
Cs-602: 273c, 298
csr32r-786: 279a, 281, 283a, 285b, 286, 288, 290b, 293, 294a, 294b, 295c, 296a, 296b, 298, 300, 301, 302b
csr32w-787: 279b, 282c, 283a, 283b, 284c, 285b, 286, 288, 289a, 289b, 290b, 293, 294a, 294b, 295c, 296a, 296b, 298, 300, 301, 302b

CsumErrs-209: 344b, 348c
CsumErrs-344: 410c, 441
CsumErrs-82: 154d, 158
CsumErrs6-137: 468, 478
Ctlr: 212b, 270b
Ctlr.active: 212b
Ctlr.next: 212b
Ctlr.pcidev: 212b
Ctlr (typedef): 270b
CtlrEtherIgbe: 278d, 307b
CtlrEtherIgbe.active: 278d
CtlrEtherIgbe.alloc: 278d
CtlrEtherIgbe.alock: 278d
CtlrEtherIgbe.cls: 278d
CtlrEtherIgbe.edev: 278d
CtlrEtherIgbe.eeprom: 278d
CtlrEtherIgbe.fcrth: 278d
CtlrEtherIgbe.fcrtl: 278d
CtlrEtherIgbe.id: 278d
CtlrEtherIgbe.im: 278d
CtlrEtherIgbe.imlock: 278d
CtlrEtherIgbe.ipcs: 278d
CtlrEtherIgbe.ixsm: 278d
CtlrEtherIgbe.lim: 278d
CtlrEtherIgbe.link: 278d
CtlrEtherIgbe.lintr: 278d
CtlrEtherIgbe.lrendez: 278d
CtlrEtherIgbe.lsleep: 278d
CtlrEtherIgbe.mii: 278d
CtlrEtherIgbe.mta: 278d
CtlrEtherIgbe.next: 278d
CtlrEtherIgbe.nic: 278d
CtlrEtherIgbe.nrb: 278d
CtlrEtherIgbe.nrd: 278d
CtlrEtherIgbe.ntd: 278d
CtlrEtherIgbe.pcidev: 278d
CtlrEtherIgbe.port: 278d
CtlrEtherIgbe.ra: 278d
CtlrEtherIgbe.rb: 278d
CtlrEtherIgbe.rdba: 278d
CtlrEtherIgbe.rdfree: 278d
CtlrEtherIgbe.rdh: 278d
CtlrEtherIgbe.rdt: 278d
CtlrEtherIgbe.rdtr: 278d
CtlrEtherIgbe.rim: 278d
CtlrEtherIgbe.rintr: 278d
CtlrEtherIgbe.rrendez: 278d
CtlrEtherIgbe.rsleep: 278d
CtlrEtherIgbe.slock: 278d

CtlrEtherIgbe.started: 278d
CtlrEtherIgbe.statistics: 278d
CtlrEtherIgbe.tb: 278d
CtlrEtherIgbe.tcpcs: 278d
CtlrEtherIgbe.tdba: 278d
CtlrEtherIgbe.tdfree: 278d
CtlrEtherIgbe.tdh: 278d
CtlrEtherIgbe.tdt: 278d
CtlrEtherIgbe.tintr: 278d
CtlrEtherIgbe.tlock: 278d
CtlrEtherIgbe.txcw: 278d
CtlrEtherIgbe.txdw: 278d
CtlrEtherIgbe (typedef): 307b
ctlrhead-378: 212c, 212e, 213b
ctlrtail-379: 212d, 212e
CtMASK-714: 276a
Ctrl-494: 271b, 285b, 294a, 294b, 296b, 301, 302b
Ctrldup-495: 271b
Ctrlext-498: 271b, 281, 296b, 301, 302b
CtSHIFT-715: 276a, 286
Curr-409: 200a, 208a, 266b
CurrEstab-336: 410c, 412c
Cxe-749: 277e
Data-376: 197b, 270a
Dcr-406: 199b, 266b
DeathTime-64: 391b
Defadvscale-330: 406, 420a
DefaultTTL: 31a, 77b
Defaultwin-66: 166a, 391b
DefByteRate-71: 166a, 391b
defmask(): 22b, 72, 179a
DefRtt-72: 166a, 391b
DEF_KAT-303: 406, 420b
DEF_MSS-300: 406, 420a, 420b
DEF_MSS6-301: 406, 420a, 420b
DEF_RTT-302: 406, 410b
Delayack-351: 410c, 449
DelayMASK-736: 277b
DelaySHIFT-737: 277b
delimchar(): 89, 224c
delroute(): 132a, 132b
Devrst-573: 272, 301
Dext-764: 278a, 288
DFLTTOS: 160, 232b, 346a, 346b, 348c, 397a, 428, 430, 474b, 475, 476, 477a, 477b, 481b
Dfr-474: 268d
DGain-70: 391b
Di-603: 273c, 298
Did-626: 274b
Dis-473: 268d

disable(): 199b, 201a, 201b
Do-604: 273c, 298
DOH: 493a, 498d
Dp8390: 196, 266a
Dp8390.data: 196
Dp8390.dummyrr: 196
Dp8390.mar: 210a
Dp8390.mref: 210a
Dp8390.nxtpkt: 196
Dp8390.port: 196
Dp8390.pstart: 196
Dp8390.pstop: 196
Dp8390.ram: 196
Dp8390.tstart: 196
Dp8390.txbusy: 196
Dp8390.width: 196
Dp8390 (typedef): 266a
Dp8390BufSz: 197b, 203, 207, 265c
dp8390read(): 197b, 199a
dp8390reset(): 197b, 199b
dp8390setea(): 197b, 200b
dp8390write(): 203, 204
Dpf-707: 275d, 289b
DPRINT-363: 145b, 403a, 515c
DroppedMsgs-89: 154d, 162b
DS: 353b, 355a
DS.buf: 353b
DS.ctlp: 353b
DS.dir: 353b
DS.local: 353b
DS.netdir: 353b
DS.proto: 353b
DS.rem: 353b
DS (typedef): 355a
DtypeCD-755: 278a
DtypeDD-756: 278a, 288
dumpnetif(): 326a
dumpoq(): 325d
dumpreseq(): 419d, 458a, 458c
DupBytes-88: 154d, 396
DupMsg-87: 154d, 396
Ea-620: 274b, 302b
eaddr (typedef): 27e
Eaddrlen: 27d, 27e, 197b, 200b, 259, 261b, 261c, 302b, 306, 321b, 323c, 323d
Ebadarp: 109b, 333c, 333c
Ebadnetctl: 191d, 388c, 388c
Ec-768: 278b
EchoReply-151: 469c, 481b
EchoReply-188: 343a, 348a, 348c

EchoReplyV6-168: 469c, 474a
EchoRequest-158: 469c
EchoRequest-192: 343a, 348c
EchoRequestV6-167: 469c, 481b
Eecd-497: 271b, 298, 300
Eepresent-607: 273c
Eerst-617: 274a, 301
Eesz256-608: 273c, 300
Eeszaddr-609: 273c, 300
eipfmt(): 186, 327
endian-16: 230a, 230a, 230b
endian-185: 390a, 390a, 390b
EOLOPT-292: 406, 425, 426
equivip4(): 90c
equivip6(): 504a
ESP: 498d
Esr: 311d, 314b
Esr1000TFD: 311d, 316b
Esr1000THD: 311d, 316b
Esr1000XFD: 316b
Esr1000XHD: 316b
Established-319: 406, 412c, 414c, 415a, 434b, 441, 455a
EstabResets-335: 410c, 441
ETARP: 240f, 360c, 362, 363
Ether: 28e, 255d
Ether.attach: 29a
Ether.ctl: 29c
Ether.ctlr: 28g
Ether.ctlrno: 28f
Ether.detach: 29a
Ether.ea: 28e
Ether.ifstat: 29c
Ether.interrupt: 29b
Ether.oq: 28e
Ether.power: 29c
Ether.shutdown: 29c
Ether.tbdf: 28h
Ether.transmit: 29b
ether2000link(): 197a
Ether (typedef): 255d
etheraddmulti(): 27f, 356c, 360a
Etherarp: 357c, 366c
Etherarp.d: 357c
Etherarp.hln: 357c
Etherarp.hrd: 357c
Etherarp.op: 357c
Etherarp.pln: 357c
Etherarp.pro: 357c
Etherarp.s: 357c

Etherarp.sha: 357c
Etherarp.spa: 357c
Etherarp.tha: 357c
Etherarp.tpa: 357c
Etherarp.type: 357c
Etherarp (typedef): 366c
etherbind(): 27f, 77c, 356c
etherbread(): 256e
etherbroadcast-181: 28a, 363
etherbwrite(): 358
etherclose(): 256c
ethercrc(): 265b
ethercreate(): 256b
etherdevtab: 65f
Etherhdr: 86a, 366c
Etherhdr.d: 86a
Etherhdr.s: 86a
Etherhdr.t: 86a
Etherhdr (typedef): 366c
ETHERHDRSIZE: 240f, 289b
etherigbelink(): 307a
etheriq(): 257b
ETHERMAXTU: 240f, 261c, 323d
ethermedium: 27f, 366a
ethermediumlink(): 366a
ETHERMINTU: 203, 240f, 261c, 323d
etheropen(): 256a
etheroq(): 258
Etherpkt: 86b, 241a
Etherpkt.d: 86b
Etherpkt.data: 86b
Etherpkt.s: 86b
Etherpkt.type: 86b
Etherpkt (typedef): 241a
etherpref2addr(): 27f, 356c, 366b
etherprobe(): 261c, 263
etherread(): 256d
etherread4(): 77c, 359
etherread6(): 77c, 506d
etherremmulti(): 27f, 356c, 360b
etherreset(): 263
Etherrock: 357a, 366c
Etherrock.achan: 357a
Etherrock.arpp: 357a
Etherrock.cchan4: 357a
Etherrock.cchan6: 357a
Etherrock.f: 357a
Etherrock.mchan4: 357a
Etherrock.mchan6: 357a

Etherrock.read4p: 357a
Etherrock.read6p: 357a
Etherrock (typedef): 366c
etherrtrace(): 257a
ethershutdown(): 264a
etherstat(): 255f
etherunbind(): 27f, 356c, 357e
etherwalk(): 255e
etherwrite(): 259
etherwstat(): 256f
etherxx-795: 28c, 255e, 255f, 256a, 256c, 256d, 256e, 256f, 259, 260a, 263, 264a
etherxx-798: 317a, 317c, 318a, 318b, 318d, 318e, 319a, 319b, 321b, 322, 323d, 325a
etime-94: 392c, 392c
ETIP4: 240f, 360c, 362, 363
ETIP6: 240f
Ett-531: 271b
Fae-469: 207, 268d
Fcah-501: 271b, 302b
Fcal-500: 271b, 302b
Fcrth-516: 271b, 302b
Fcrtl-515: 271b, 302b
Fct-502: 271b, 302b
Fcttv-508: 271b, 302b
Fd-595: 273b, 285b, 302b
FH: 493a, 493b, 498d
Fifo-387: 266b
FIN-291: 406, 428, 441, 449, 453, 459b
findfield(): 74, 76b
findipifc(): 113d, 113e, 420a, 508b
findlocalip(): 140b, 149a, 329c, 379, 420b, 514a
findprimaryipv4(): 378e, 379
findprimaryipv6(): 379, 511c
Finwait1-320: 406, 414c, 441
Finwait2-321: 406, 441, 449, 453, 462d
flags-254: 191d, 388b
Fo-470: 207, 268d
FORCE-308: 406, 416a, 427, 441, 449, 455c, 457b
Forwarding: 31a, 80a
ForwDatagrams: 31a, 104e, 490
Fpd-738: 277b, 282c, 289b
FragCreates: 31a, 95c, 486e
FragFails: 31a, 95c, 486e
Fraghdr6: 499b, 500
Fraghdr6.id: 499b
Fraghdr6.nexthdr: 499b
Fraghdr6.offsetRM: 499b
Fraghdr6.res: 499b
Fraghdr6 (typedef): 500
Fragment4: 234b, 234b

Fragment4 (typedef): 234b
Fragment6: 234b, 234b
Fragment6 (typedef): 234b
FragOKs: 31a, 95c, 486e
Frcdplx-568: 272, 285b, 296b
Frcspd-567: 272, 285b, 296b, 302b
freeiplink-105: 377a, 377c
freeipself-106: 377b, 378a
freeroute(): 116a, 117, 118b, 131a, 507c, 508a
Fs: 234b, 234b
Fs (typedef): 234b
fslock: 30a, 40b
Fsnewcall(): 142, 144, 158, 434b
Fsproto(): 67, 68a, 134, 150, 170, 351b, 484a
Fsprotoclone(): 52a, 52b, 144
Fsrcvpcol(): 98, 99d, 490
Fsrcvpcolx(): 331a, 348c, 481b
Fsstdannounce(): 58b, 140a, 156d, 345c, 414b
Fsstdbind(): 61b, 61c
Fsstdconnect(): 56b, 139c, 156c, 345a, 413a
Ft0-439: 267c, 267c
Ft1-440: 267c, 267c
Ft1WORD-441: 267c
Ft2WORD-442: 267c
Ft4WORD-443: 199b, 267c
Ft6WORD-444: 267c
Fu-458: 205b, 268b
gbemedium: 356c, 366a
getcurr(): 207, 208a
getreseq(): 441, 459a
GGP: 498d
globalv6-112: 510b, 511a, 511c
Global_scop: 498e
Gorcl-546: 271b, 281
Gotcl-547: 271b, 281
goticmpkt(): 347c, 348c
goticmpkt6(): 473b, 481b
Gpi0-649: 274d
Gpi1-650: 274d
Gpi2-651: 274d
Gpi3-652: 274d
Gpien-611: 274a
Gran-728: 276b, 286
hash(): 252b, 252c, 253
haship-119: 109b, 333d, 334b, 335, 336, 338
hashipa-103: 125e, 127b, 128a, 128b, 378d
hashipa-362: 431a, 431b, 433, 434b
HBH: 493b, 498d
Hdr: 268e, 269

Hdr.len0: 268e
Hdr.len1: 268e
Hdr.next: 268e
Hdr.status: 268e
Hdr (typedef): 269
HdrlenMASK-773: 278b
HdrlenSHIFT-774: 278b
HlenErrs-211: 344b, 348c
HlenErrs-345: 410c, 441
HlenErrs-83: 154d, 158
HlenErrs6-139: 468, 478
hnputl(): 228b
hnputs(): 228c
hnputv(): 228a
HoplimErrs6-140: 468, 478
HOP_LIMIT: 474b, 475, 476, 477a, 477b, 478, 498e
Hostparams: 234b
Hostparams (typedef): 234b
HthreshMASK-724: 276b
HthreshSHIFT-725: 276b, 286, 289b
htontcp4(): 424, 428, 429, 430, 449, 453
htontcp6(): 422, 428, 429, 430, 449, 453
i82540em-482: 271a, 285b, 286, 289b, 296b, 300, 301, 305
i82540eplp-483: 271a, 285b, 286, 289b, 296b, 300, 301, 305
i82541ei-488: 271a, 286, 296b, 300, 305
i82541gi-489: 271a, 285b, 286, 289b, 296b, 300, 301, 302b, 305
i82541gi2-490: 271a, 285b, 286, 289b, 296b, 300, 301, 305
i82541pi-491: 271a, 285b, 286, 289b, 296b, 300, 301, 305
i82542-477: 271a
i82543gc-478: 271a, 285b, 286, 296b, 302b, 305
i82543mdior(): 294a, 295a
i82543mdiow(): 294b, 295a, 295b
i82543miimir(): 295a, 296b
i82543miimiw(): 295b, 296b
i82544ei-479: 271a, 285b, 286, 296b, 305
i82544eif-480: 271a, 285b, 286, 296b, 305
i82544gc-481: 271a, 286, 296b, 305
i82545em-484: 271a, 286, 289b, 296b, 300, 301, 305
i82545gmc-485: 271a, 286, 289b, 296b, 300, 301, 305
i82546eb-493: 271a, 286, 289b, 296b, 300, 301, 302b, 305
i82546gb-492: 271a, 286, 289b, 296b, 300, 301, 302b, 305
i82547ei-486: 271a, 286, 296b, 300, 305
i82547gi-487: 271a, 285b, 286, 289b, 296b, 300, 301, 305
ICMP: 498d
Icmp: 342, 342
Icmp.cksum: 342
Icmp.code: 342
Icmp.data: 342
Icmp.dst: 342

Icmp.frag: 342
Icmp.icmpid: 342
Icmp.id: 342
Icmp.ipcksum: 342
Icmp.length: 342
Icmp.proto: 342
Icmp.seq: 342
Icmp.src: 342
Icmp.tos: 342
Icmp.ttl: 342
Icmp.type: 342
Icmp.vihl: 342
icmp6init(): 484a
Icmp6_adr_unreach: 340c, 498e
Icmp6_ad_prohib: 498e
Icmp6_gress_src_fail: 498e
Icmp6_no_route: 498e
Icmp6_out_src_scope: 498e
Icmp6_port_unreach: 498e, 515d
Icmp6_rej_route: 498e
Icmp6_unknown: 481b, 498e
Icmp (typedef): 342
icmpadvise(): 350, 351b
icmpadvise6(): 472c, 484a
icmpannounce(): 345c, 351b, 484a
icmpcantfrag(): 95c, 347b
Icmpcb6: 470g, 470g
Icmpcb6.headers: 470g
Icmpcb6 (typedef): 470g
icmpclose(): 345d, 351b, 484a
IcmpCodeErrs6-141: 468, 478
icmpconnect(): 345a, 351b, 484a
icmpcreate(): 344e, 351b
icmpcreate6(): 471c, 484a
icmpctl6(): 473a, 484a
icmphostunr(): 340c, 476, 515d
icmpinit(): 351b
icmpiput(): 348c, 351b
icmpiput6(): 481b, 484a
icmpkick(): 344e, 345e
icmpkick6(): 471c, 472d
icmpna(): 475, 481b
icmpnames: 343c, 348c, 351a
icmpnames6: 470h, 483
icmpnoconv(): 146b, 347a
icmpns(): 340c, 361, 474b, 517d
icmppkttoobig6(): 477b, 486e
Icmppriv: 344d, 352
Icmppriv.in: 344d

Icmppriv.out: 344d
Icmppriv.stats: 344d
Icmppriv6: 470f, 470f
Icmppriv6.in: 470f
Icmppriv6.out: 470f
Icmppriv6.stats: 470f
Icmppriv6 (typedef): 470f
Icmppriv (typedef): 352
icmpstate(): 345b, 351b, 484a
icmpstats(): 351a, 351b
icmpstats6(): 483, 484a
icmpttlexceeded(): 104e, 346a
icmpttlexceeded6(): 477a, 490
icmpunreachable(): 346b, 347a, 347b
ICMPv6: 472a, 474b, 475, 476, 477a, 477b, 478, 484a, 490, 498d
ICMP_HDRSIZE-205: 344a, 345e, 346a, 346b, 348c
ICMP_IPSIZE-204: 344a, 345e, 346a, 346b, 348a, 348c
ICMP_USEAD6-147: 469a, 472d
Icr-503: 271b, 293, 301
Ics-504: 271b
Icw1-623: 274b, 302b
Icw2-628: 274b, 302b
Ide-766: 278a
Idle: 34b, 50c, 52b, 55c, 57c, 139b
IDRP: 498d
Ifcs-760: 278a, 288
igbeattach(): 292, 306
igbectl(): 282c, 306
igbectlmsg-794: 282b, 282c
igbectlrhead-788: 279c, 305, 306
igbectlrtail-789: 279d, 305
igbedetach(): 301, 302a, 302b
igbeifstat(): 281, 306
igbeim(): 284c, 285b, 288, 290b
igbeinterrupt(): 293, 306
igbelim(): 285a, 285b
igbelproc(): 285b, 292
igbemii(): 296b, 302b
igbemiimir(): 295c, 296b
igbemiimiw(): 296a, 296b
igbemulticast(): 283b, 306
igbepci(): 305, 306
igbepnp(): 306, 307a
igbepromiscuous(): 283a, 306
igberballoc(): 284a, 289a, 292
igberbfree(): 284b, 292
igberblock-790: 280a, 284a, 284b
igberbpool-791: 280b, 284a, 284b
igbereplenish(): 289a, 289b, 290b

igbereset(): 302b, 305
igberim(): 290a, 290b
igberproc(): 290b, 292
igberxinit(): 289b, 290b
igbeshutdown(): 302a, 306
igbetransmit(): 288, 293, 306
igbetxinit(): 286, 292
IGMP: 498d
IGRP: 498d
Iixsm-771: 278b
Ilack-55: 152d, 162b
ilackq(): 157, 392d
ilackto(): 162b, 393
iladvise(): 150, 400b
ilannounce(): 150, 156d
ilbackoff(): 394, 398a
Ilcb: 153, 401b
Ilcb.ackq: 153
Ilcb.acksent: 153
Ilcb.acktime: 153
Ilcb.conv: 153
Ilcb.delay: 153
Ilcb.fasttimeout: 153
Ilcb.lastrecv: 153
Ilcb.lastxmit: 153
Ilcb.maxrtt: 153
Ilcb.mdev: 153
Ilcb.next: 153
Ilcb.outo: 153
Ilcb.outoforder: 153
Ilcb.qt: 153
Ilcb.qtx: 153
Ilcb.querytime: 153
Ilcb.rate: 153
Ilcb.recvd: 153
Ilcb.rexmit: 153
Ilcb.rstart: 153
Ilcb.rttack: 153
Ilcb.rttlen: 153
Ilcb.rttstart: 153
Ilcb.rxquery: 153
Ilcb.rxtot: 153
Ilcb.start: 153
Ilcb.state: 153
Ilcb.timeout: 153
Ilcb.unacked: 153
Ilcb.unackedbytes: 153
Ilcb.unackedtail: 153
Ilcb.window: 153

Ilcb (typedef): 401b
ilcbinit(): 158, 165, 166a
ilcksum: 157, 160, 167b, 167b, 397a
ilclose(): 150, 166b
Ilclose-58: 152d, 158, 162b, 166b, 397a
Ilclosed-45: 152a, 152c, 157, 162b, 165, 166b, 167a
Ilclosing-50: 152a, 157, 162b, 166b
ilconnect(): 150, 156c
ilcreate(): 150, 155c
Ildata-53: 152d, 157, 162b
Ildataquery-54: 152d, 162b, 394
Ilestablished-48: 152a, 162b, 166b, 395b
ilfreeq(): 162b, 166b, 400a
ilhangup(): 162b, 395a, 400b
Ilhdr: 155a, 401b
Ilhdr.cksum: 155a
Ilhdr.dst: 155a
Ilhdr.frag: 155a
Ilhdr.id: 155a
Ilhdr.ilack: 155a
Ilhdr.ildst: 155a
Ilhdr.ilid: 155a
Ilhdr.illen: 155a
Ilhdr.ilspec: 155a
Ilhdr.ilsrc: 155a
Ilhdr.ilsum: 155a
Ilhdr.iltype: 155a
Ilhdr.length: 155a
Ilhdr.proto: 155a
Ilhdr.src: 155a
Ilhdr.tos: 155a
Ilhdr.ttl: 155a
Ilhdr.vihl: 155a
Ilhdr (typedef): 401b
ilinit(): 150
ilinuse(): 150, 152c
iliput(): 150, 158
ilkick(): 155c, 157
Illistening-49: 152a, 157, 158, 162b, 165, 166b
illocalclose(): 162b, 166b, 167a, 395a
ilnextqt(): 394, 401a
Ilopening-51: 152a
Ilos-561: 272, 285b, 302b
iloutoforder(): 162b, 396
Ilpriv: 154a, 401b
Ilpriv.ackprocstarted: 154a
Ilpriv.apl: 154a
Ilpriv.csumerr: 154b
Ilpriv.dup: 154a

Ilpriv.dupb: 154a
Ilpriv.hlenerr: 154b
Ilpriv.ht: 154a
Ilpriv.lenerr: 154b
Ilpriv.order: 154b
Ilpriv.rexmit: 154b
Ilpriv.stats: 154c
Ilpriv (typedef): 401b
ilprocess(): 158, 162a
ilpullup(): 162b, 395b
Ilquery-56: 152d, 162b
ilreject(): 158, 397a
ilrexmit(): 162b, 394
ilrttcalc(): 392e, 393
ilsendctl(): 158, 160, 162b, 165, 166b
ilsettimeout(): 157, 162b, 166a, 166b, 393, 397b
ilstart(): 156c, 156d, 165
ilstate(): 150, 168b
Ilstate-57: 152d, 162b
ilstates: 152b, 168b
Ilsync-52: 152d, 158, 162b, 165
Ilsyncee-47: 152a, 158, 162b, 166b
Ilsyncer-46: 152a, 162b, 165, 166b, 395a, 400b
Iltickms-60: 165, 391b, 391b, 398a
iltype: 152e, 158
ilxstats(): 150, 168a
IL_CONNECT-79: 156b, 156c, 165
IL_HDRSIZE-76: 155b, 157, 158, 160, 392e, 395b, 397a
IL_IPSIZE-75: 155b, 157, 158, 160, 392e, 394, 395b, 397a
IL_LISTEN-78: 156b, 156d, 165
il_state: 152a
il_stat: 154d
Imc-506: 271b, 293, 301
Imr-407: 199b, 201c, 205b, 266b
Ims-505: 271b, 284c, 293
InAddrErrors: 31a
incoming(): 62g, 328b
InDelivers: 31a, 98, 490
InDiscards: 31a, 98, 490
InErrors-207: 344b, 348c
InErrors6-135: 468, 478
InErrs-342: 410c, 441
InfoReply-164: 469c
InfoReply-198: 343a
InfoRequest-163: 469c
InfoRequest-197: 343a
InHdrErrors: 31a, 99a, 99c, 104e, 490
initfrag(): 42a, 42b
initialwindow(): 434a, 434b, 457b

inittcpctl(): 420b, 421
inittimescale(): 150, 151b
InMsgs-206: 344b, 348c
InMsgs-80: 154d, 162b
InMsgs6-134: 468
InParmProblem-160: 469c
InParmProblem-194: 343a
InReceives: 31a, 98, 490
InSegs-337: 410c, 441
interrupt(): 199b, 205b
InUnknownProtos: 31a, 98, 490
inwindow(): 457a, 459b
IP: 234b, 234b
ip1gen(): 45c, 46b, 46c
ip2gen(): 46d, 47a, 47b
ip3gen(): 47c, 47d, 48a
IP4HDR: 95c, 100c, 103a, 232b
Ip4hdr: 234b, 234b
Ip4hdr (typedef): 234b
ip4reassemble(): 100b, 100c, 104e
IP6FHDR-259: 486a, 486e
IP6HDR: 472a, 472d, 478, 486e, 490, 493a, 493b, 498e
Ip6hdr: 499a, 500
Ip6hdr.payload: 499a
Ip6hdr (typedef): 500
IP (typedef): 234b
ipaddr (typedef): 20d
IPaddrlen: 20c, 20d, 22d, 71c, 72, 82b, 89, 113e, 118a, 121c, 127b, 136c, 149a, 149c, 179a, 224d, 372e, 470b, 470d, 472d, 474a, 474b, 475, 476, 477a, 477b, 481b, 490, 492a, 501a, 504a, 504b, 504c, 504d, 504f, 504g, 505b, 505c, 507c, 508b, 512c, 513b
IPallbits: 127b, 128b, 182d, 227d, 501b
ipattach(): 40a, 44a
IPaux: 234b, 234b
IPaux (typedef): 234b
ipbread(): 44a, 329b
ipbroadcast-180: 356b, 363
ipbwrite(): 44a, 330b
ipcharok(): 89, 224b
ipclose(): 44a, 50a
ipcmp: 23d, 71c, 73b, 76c, 113e, 125e, 127b, 128b, 136d, 140b, 143, 182d, 191d, 329d, 338, 347c, 350, 362, 363, 378d, 379, 381a, 381b, 382a, 382b, 383c, 400b, 404, 420b, 429, 431b, 433, 434b, 461, 472c, 473b, 478, 481a, 481b, 501b, 511a, 514a, 514b, 516a, 516f, 517f
ipcreate(): 44a, 44b
Ipcs-744: 277d, 290b
ipcsum(): 91, 95a, 95c, 99a
ipdevtab: 44a
Ipe-751: 277e
ipfindmedium(): 69, 70c, 109b
ipforme(): 59, 98, 125e, 148b, 149a, 346b, 365b, 490

Ipfrag: 234b
Ipfrag (typedef): 234b
ipfragallo4(): 92b, 100c
ipfragallo6(): 492b
ipfragfree4(): 92a, 92b, 100c
ipfragfree6(): 492a, 492b
IPFRAGSZ: 100c, 232d
ipfs: 29f, 40b, 45b, 46c, 48a, 49a, 50a, 51a, 51c, 328c, 329b, 330b
ipgen(): 45a, 45b, 51b, 328a
ipgetfs(): 40a, 40b
Iphash: 234b, 234b
iphash(): 136c, 136d, 137a, 143
Iphash (typedef): 234b
Ipht: 234b, 234b
Ipht (typedef): 234b
iphtadd(): 136d, 139c, 140a, 142, 158, 165, 421, 434b
iphtlook(): 142, 143, 158, 441
iphtrem(): 137a, 139b, 167a, 419d
IPICMP: 469d, 484b
IPICMP.payload: 469d
IPICMP (typedef): 484b
IPICMPSZ-175: 470a, 472d, 476, 477a, 477b, 478, 481b
Ipifc: 234b, 234b
Ipifc (typedef): 234b
ipifcadd(): 71b, 71c, 376c, 512c, 517a
ipifcadd6(): 512c, 517b
ipifcaddmulti(): 180c, 382a
ipifcaddroute(): 114, 133b, 507c
ipifcbind(): 67, 69
ipifcclose(): 67, 84b
ipifcconnect(): 67, 376c
ipifccreate(): 67, 68d
ipifcctl(): 67, 71a
ipifcinit(): 67
ipifcinuse(): 67, 376a
ipifcjoinmulti(): 180e, 383a
ipifckick(): 68d, 376b
ipifcleavemulti(): 180f, 383b
ipifclocal(): 67, 375d
ipifcra6(): 510a, 517c
ipifcregisterproxy(): 182e, 383c
ipifcrem(): 82a, 82b
ipifcremmulti(): 180d, 181c, 382b
ipifcremroute(): 131a, 133c, 508a
ipifcsetmtu(): 80c, 80d
ipifcstats(): 67, 77a
ipifcunbind(): 83a, 83b, 84b
IPINIP: 498d
ipiput4(): 98, 218b, 359, 368d

ipiput6(): 476, 490, 506d, 515a
ipismulticast(): 180c, 180d, 334b, 365a, 381c
Iplifc: 234b, 234b
Iplifc (typedef): 234b
Iplink: 234b, 234b
Iplink (typedef): 234b
iplinkfree(): 128b, 377c
IPllen: 121c, 232b, 507c, 508a, 508b, 519a
iplocalonifc(): 363, 381a, 481b
iplong (typedef): 23a
IPmatchaddr: 136b, 136d, 143
IPmatchany: 136b, 136d, 143
IPmatchexact: 136b, 136d, 143
IPmatchpa: 136b, 136d, 143
IPmatchport: 136b, 136d, 143
Ipmcast: 180a, 384b
Ipmcast.ia: 180a
Ipmcast.ma: 180a
Ipmcast.next: 180a
Ipmcast (typedef): 384b
ipmove: 23c, 52b, 59, 71c, 76c, 113e, 127b, 139b, 144, 149a, 149c, 167a, 345d, 378e, 379, 382a, 404, 420b, 428, 430, 431b, 434b, 441, 461, 472d, 474a, 511c, 512a, 512b, 514a, 515c, 516b
Ipmulti: 234b, 234b
Ipmulti (typedef): 234b
IPnoaddr: 23b, 52b, 59, 71c, 76c, 127b, 128b, 136d, 139b, 140b, 143, 167a, 191d, 345d, 362, 420b, 429, 501a, 511a, 514a, 514b, 516a, 516f
Ipofl-733: 276c
ipopen(): 44a, 49a
ipoput4(): 93, 104e, 140b, 157, 160, 345e, 346a, 346b, 348c, 394, 397a, 428, 429, 430, 449, 453
ipoput6(): 428, 429, 430, 449, 453, 472d, 474b, 475, 476, 477a, 477b, 481b, 486e, 490, 514a
ipproxyifc(): 363, 381b, 481a
ipread(): 44a, 51a
ipremove(): 44a, 44c
ipreset(): 44a, 327
iproute(): 123b, 125a
iprouting(): 79, 80a
Ips-618: 274a, 302b
Ipself: 126c, 234b
Ipself.a: 126c
Ipself.expire: 126c
Ipself.hnext: 126c
Ipself.link: 126c
Ipself.next: 126c
Ipself.ref: 126c
Ipself.type: 126c
Ipself (typedef): 234b
ipselffree(): 128b, 378a
Ipselftab: 126a, 234b
Ipselftab.acceptall: 126a

Ipselftab.hash: 126a
Ipselftab.inited: 126a
Ipselftab (typedef): 234b
ipselftabread(): 130c, 130d
ipstat(): 44a, 328a
ipstats(): 77a, 77b
iptentative(): 378d, 486e, 490
ipv4 (typedef): 20b
IPv4addrlen: 20a, 20b, 118a, 121c, 224d, 380
IPv4allrouter: 227c
IPv4allsys: 227b
IPv4bcast: 179a, 179b
ipv4local(): 360c, 380
IPv4off: 21a, 21b, 22b, 71c, 89, 109b, 113d, 121c, 123b, 125a, 127b, 128b, 132b, 224d, 338, 360c, 362, 379, 380, 381c, 508b, 514b, 516a, 516f
ipv4or6 (typedef): 21e
ipv4p (typedef): 21f
ipv62smcast(): 372e, 383c, 474b, 517f
ipv6anylocal(): 340c, 361, 474a, 476, 477a, 477b, 512b
IPV6CLASS-260: 486b, 490
ipv6local(): 481b, 512a
ipwalk(): 44a, 45a
ipwalkroutes(): 120d, 122b
ipwrite(): 44a, 51c
ipwstat(): 44a, 328c
IP_DF: 95c, 100c, 232b
IP_HLEN4: 93, 94b, 99c
IP_ICMPPROTO-203: 344a, 345e, 346a, 346b, 351b
IP_ILPROTO-77: 150, 151a, 157, 160, 397a
IP_MAX: 97a, 97b, 486e
IP_MF: 95c, 100b, 100c, 104e, 232b
IP_TCPPROTO-269: 170, 406, 420b, 428, 430, 434b
IP_UDPPROTO-371: 134, 135a, 140b, 514a, 515c
IP_VER4: 93, 94a, 99c, 140b, 157, 160, 345e, 346a, 346b, 348a, 394, 397a, 428, 429, 430, 441, 449, 453, 461, 515a
IP_VER6: 232b, 404, 422, 428, 429, 430, 449, 453, 486e, 490, 514a, 516c
ip_init(): 40b, 42a
ip_init_6(): 502b, 505e
ip_version: 21g
islinklocal: 478, 490, 498a, 511a, 512c
ISO_IP: 498d
ISO_TP4: 498d
Isprefix-3: 186, 226c, 226d
Isprefix-97: 372d
Isr-388: 199b, 201a, 201c, 204, 205b, 208b, 209, 266b
issmcast: 478, 498c
isv4(): 21b, 22b, 71c, 109b, 125a, 127b, 128b, 334b, 337b, 380, 381c, 383c, 453, 511a, 512a, 512b, 516k
isv6mcast: 474a, 476, 477a, 477b, 478, 490, 497b, 511a
Itxsm-772: 278b

Ixsm-741: 277d, 290b
L-361: 417a, 417b
Lanid-578: 273a, 302b
Las-436: 267c
Last_ack-324: 406, 414c, 441
later(): 157, 398c
LB: 217d, 356a
LB.f: 217d
LB.q: 217d
LB.readp: 217d
Lb0-446: 268a, 268a
Lb1-447: 268a, 268a
LB (typedef): 356a
LbmMASK-683: 275d
LbmMII-686: 275d
LbmOFF-684: 275d
LbmTBI-685: 275d
LbmXCVR-687: 275d
Lc-769: 278b
lcmp(): 518e, 519a
LenErrs-210: 344b, 348c
LenErrs-346: 410c, 441
LenErrs-84: 154d, 158
LenErrs6-138: 468, 478, 481b
LenMASK-753: 278a, 288
LenSHIFT-754: 278a, 288
LHTMASK-328: 406
Limbo: 410a, 464
limbo(): 431b, 441
Limbo.irs: 410a
Limbo.iss: 410a
Limbo.laddr: 410a
Limbo.lastsend: 410a
Limbo.lport: 410a
Limbo.mss: 410a
Limbo.next: 410a
Limbo.raddr: 410a
Limbo.rcvscale: 410a
Limbo.rexmits: 410a
Limbo.rport: 410a
Limbo.sndscale: 410a
Limbo.version: 410a
Limbo (typedef): 464
limborexmit(): 418, 432
limborst(): 433, 441
linklocalv6-111: 510b, 511a
Link_local_scop: 490, 498e, 511a
Listen-316: 406, 414c, 421, 441, 449
LITTLE-18: 230c, 230d

LITTLE-187: 390c, 390d
localclose(): 414c, 415a, 419d, 429, 441, 449, 454b, 456, 461, 462d
LOGAGAIN-313: 406, 420b, 437e, 438, 463a
LogAGain-67: 166a, 168b, 391b, 391b, 392e, 397b, 398a
Logcompress: 233d, 388b
Logconv-238: 35a, 35a
LOGDGAIN-314: 406, 437e, 438
LogDGain-69: 166a, 168b, 391b, 391b, 392e, 397b, 398a
Logesp: 233d, 388b
Logfs: 233d, 388b
Loggre: 233d, 388b
Logicmp: 233d, 346a, 346b, 348c, 388b, 474b, 475, 476, 477a, 477b, 478
Logigmp: 233d
Logip: 95c, 97b, 97c, 99a, 99c, 103b, 233d, 388b, 486e, 490
Logipmsg: 233d, 388b
Logppp: 233d, 388b
Logproto-241: 35a, 35a
logreseq(): 458b, 458c
Logrudp: 233d
Logrudpmsg: 233d
Logtcp: 233d, 388b, 415b, 428, 434b, 438, 441, 458b, 458c, 463b
Logtcprxmt: 233d, 388b, 456
Logtcpwin: 233d, 388b, 438, 456
Logtype-236: 35a, 35a
Logudp: 145b, 146a, 146b, 147a, 233d, 388b, 515c
Logudpmsg: 142, 233d, 388b
looknode(): 131a, 131b, 508a
lookupcmd(): 191d, 282c
loopbackbind(): 217a, 217e
loopbackbwrite(): 217a, 218c
loopbackmask-2: 76c, 226a
loopbackmedium: 217a, 217c
loopbackmediumlink(): 217c
loopbacknet-1: 76c, 225c
loopbackread(): 217e, 218b
loopbackunbind(): 217a, 218a
LpbkENDEC-450: 268a
LpbkEXTERNAL-451: 268a
LpbkNIC-449: 199b, 209, 268a
LpbkNORMAL-448: 201c, 209, 268a
Lpe-682: 275d
lportinuse(): 58f, 330a
Lroot: 39b
Lrst-558: 272, 302b
Ls-437: 199b, 267c
Lsc-642: 274d, 285b, 293
Lspeed10-583: 273a
Lspeed100-584: 273a
Lspeed1000-585: 273a

LspeedMASK-581: 273a
LspeedSHIFT-582: 273a
LthreshMASK-729: 276b
LthreshSHIFT-730: 276b
Lu-577: 273a
m2p-251: 48c, 49a
MACLen: 108d, 109b, 232b
Manc-555: 271b, 301
Mar0-410: 211c, 266b
Maskconv-239: 35a
maskip(): 22d, 72, 76c, 82b, 113e, 379, 381b, 383c
Maskproto-242: 35a
Masktype-237: 35a
matchtoken(): 246c, 251a
matchtype: 136b
MAXBACKMS-285: 406, 438, 456
MaxConn-331: 170, 410c
MaxEther: 28c, 28d, 260b, 261a, 261c, 263, 264a, 317a, 323a, 323b, 323d, 325a
Maxhdrtype: 498d
Maxincall: 144, 232b
Maxlimbo-326: 406, 431b
Maxmedia-98: 27a, 27b, 67, 374
Maxpath: 77c, 232b, 353c, 353d
Maxproto: 29e, 68a
Maxqscale-329: 406, 463b
MaxRexmit-65: 165, 391b
Maxrq-73: 155c, 156a, 162b
Maxstring-96: 353a, 353b, 354
MaxTimeout-62: 391b, 397b, 398a
MAXTTL: 52b, 69, 77b, 81f, 81g, 160, 346a, 346b, 348c, 397a, 428, 430, 474b, 475, 476, 477a, 477b, 481b, 502b
Maxtu-263: 217a, 217b, 218b
Maxtype-201: 343a, 343c, 344d, 345e, 348c, 351a
Maxtype6-174: 469c, 470f, 470h, 472d, 481b, 483
MAX_MULTICAST_SOLICIT: 334b, 498e
MAX_TIME-283: 406, 420b
Mdac-647: 274d
Mdc-655: 275a, 294a, 294b
Mdco-656: 275a, 294a, 294b
Mdd-653: 275a, 294a, 294b
Mddo-654: 275a, 294a, 294b
Mdic-499: 271b, 295c, 296a
MDIdMASK-629: 274c, 296a
MDIdSHIFT-630: 274c
MDIe-639: 274c, 295c, 296a
MDIie-638: 274c
MDIpMASK-633: 274c
MDIpSHIFT-634: 274c, 295c, 296a
MDIready-637: 274c, 295c, 296a
MDIrMASK-631: 274c

MDIrop-636: 274c, 295c
MDIrSHIFT-632: 274c, 295c, 296a
MDIwop-635: 274c, 296a
Mdr-657: 275a, 296b
Mdro-658: 275a, 296b
media: 27a, 27c, 70c
Medium: 234b, 234b
Medium (typedef): 234b
mib_two_counters: 31a
Mii: 316c, 316e
mii(): 296b, 310
Mii.ctlr: 316c
Mii.curphy: 316c
Mii.mask: 316c
Mii.mir: 316c
Mii.miw: 316c
Mii.nphy: 316c
Mii.phy: 316c
Mii (typedef): 316e
miiane(): 296b, 311d
miimir(): 281, 296b, 311a
miimiw(): 296b, 311b
MiiPhy: 316d, 316e
MiiPhy.anar: 316d
MiiPhy.fc: 316d
MiiPhy.fd: 316d
MiiPhy.link: 316d
MiiPhy.mii: 316d
MiiPhy.mscr: 316d
MiiPhy.oui: 316d
MiiPhy.phyno: 316d
MiiPhy.rfc: 316d
MiiPhy.speed: 316d
MiiPhy.tfc: 316d
MiiPhy (typedef): 316e
miireset(): 296b, 311c
miistatus(): 285b, 313
MinAdvise-202: 343b, 348c
mkechoreply(): 348a, 348c
mkechoreply6(): 474a, 481b
Mo43b32-696: 275d
Mo45b34-695: 275d
Mo46b35-694: 275d
Mo47b36-693: 275d, 283a
mode: 156b
MoMASK-692: 275d, 283a
Mon-466: 199b, 268c
Mpa-471: 268d
Mpe-681: 275d, 283a, 289b

Mscr: 311d, 314b
Mscr1000TFD: 311d, 313, 315d
Mscr1000THD: 311d, 313, 315d
MSL2-298: 406, 441
MSPTICK-299: 406, 418, 420b, 438, 441, 454a, 455a, 456, 463a
Mss-332: 410c, 420b, 427, 434b, 457b
MssMASK-780: 278b
MSSOPT-294: 406, 422, 424, 425, 426
Mssr: 313, 314b
Mssr1000TFD: 313, 316a
Mssr1000THD: 313, 316a
MssSHIFT-781: 278b
MSS_LENGTH-295: 406, 422, 424, 425, 426
Mta-552: 271b, 283b, 302b
MTU_OPTION: 498e
Mtxckok-586: 273a
multicast(): 199b, 210c
multicastarp(): 358, 365b
multicastea(): 360a, 360b, 365a, 365b
myetheraddr(): 85b
myipaddr(): 76c
N2ndqid: 240a, 242
N3rdqid: 240a, 242
Naddrqid: 240a, 242, 245
nbmsg-184: 77c, 357d, 357d
NbrAdvert-172: 469c, 475, 478, 481b
NbrSolicit-171: 469c, 474b, 478, 481b
NCACHE-101: 374
NCACHE-116: 106b, 108d, 109b, 333a, 334b
Nchans: 134, 135b, 151d
Ncloneqid: 240a, 242, 244b
Ncr-386: 266b
Nctlqid: 240a, 242, 244b, 245, 246c, 248b
Ndataqid: 240a, 242, 244b, 245, 246a, 248b, 259, 260a, 321b, 322
ndbwrite(): 182c, 331b
NdiscC: 470b
NdiscC.payload: 470b
NdiscC.target: 470b
NdiscC (typedef): 484b
NDISCSZ-176: 470c, 474b
Ndpkt: 470d, 484b
Ndpkt.lnaddr: 470d
Ndpkt.olen: 470d
Ndpkt.otype: 470d
Ndpkt.payload: 470d
Ndpkt.target: 470d
Ndpkt (typedef): 484b
NDPKTSZ-177: 470e, 474b, 475
ne2000match(): 212e, 213b

ne2000pci-380: 212e, 213a
ne2000pnp(): 197b, 212e
ne2000reset(): 197a, 197b
Netaddr: 240e, 241a
Netaddr.addr: 240e
Netaddr.hnext: 240e
Netaddr.next: 240e
Netaddr.ref: 240e
Netaddr (typedef): 241a
Netfile: 37a, 241a
Netfile.bridge: 37a
Netfile.headersonly: 37a
Netfile.in: 37a
Netfile.inuse: 37a
Netfile.maddr: 37a
Netfile.mode: 37a
Netfile.nmaddr: 37a
Netfile.owner: 37a
Netfile.prom: 37a
Netfile.scan: 37a
Netfile.type: 37a
Netfile (typedef): 241a
NETID: 240c, 242, 244b, 245, 246a, 246c, 247, 248b
Netif: 35f, 241a
Netif.addr: 35f
Netif.alen: 35f
Netif.all: 35f
Netif.arg: 35f
Netif.bcast: 35f
Netif.buffs: 36b
Netif.crcs: 36b
Netif.f: 35f
Netif.frames: 36b
Netif.hwmtu: 36a
Netif.inpackets: 36b
Netif.limit: 35f
Netif.link: 35f
Netif.maddr: 181d
Netif.maxmtu: 35f
Netif.mbps: 35f
Netif.mhash: 181d
Netif.minmtu: 35f
Netif.misses: 36b
Netif.mtu: 35f
Netif.multicast: 181e
Netif.name: 35f
Netif.nfile: 35f
Netif.nmaddr: 181d
Netif.oerrs: 36b

Netif.outpackets: 36b
Netif.overflows: 36b
Netif.prom: 35f
Netif.promiscuous: 36a
Netif.scan: 35f
Netif.scanbs: 36a
Netif.soverflows: 36b
Netif (typedef): 241a
netifbread(): 246a, 256e, 319a
netifclose(): 248b, 256c, 318d
netifgen(): 242, 244a, 245, 248a
netifinit(): 241b, 261c, 323d
netifopen(): 244b, 256a, 318b
netifread(): 245, 256d, 318e
netifstat(): 248a, 255f, 318a
netifwalk(): 244a, 255e, 317c
netifwrite(): 246c, 259, 321b
netifwstat(): 247, 256f, 319b
netlock: 249a, 249b
Netlog: 188b, 234b
netlog(): 95c, 97b, 97c, 99a, 99c, 103b, 142, 145b, 146a, 146b, 147a, 346a, 346b, 348c, 389a, 415b, 428, 434b, 438, 441, 456, 458b, 458c, 463b, 474b, 475, 476, 477a, 477b, 478, 486e, 490, 515c
Netlog.buf: 188b
Netlog.end: 188b
Netlog.iponly: 188b
Netlog.iponlyset: 188b
Netlog.len: 188b
Netlog.logmask: 188b
Netlog.opens: 188b
Netlog.rptr: 188b
Netlog (typedef): 234b
netlogclose(): 189g, 388d
netlogctl(): 189i, 191d
Netlogflag: 388a, 388a
Netlogflag.mask: 388a
Netlogflag.name: 388a
Netlogflag (typedef): 388a
netloginit(): 40b, 189c
netlogopen(): 189f, 190a
netlogread(): 189h, 190b
netlogready(): 190b, 191a
netmulti(): 246c, 248b, 253
netown(): 244b, 247, 249b, 250
NETQID: 240d, 242, 244b
NETTYPE: 240b, 242, 244b, 245, 246a, 246c, 248b, 256d, 259, 260a, 318e, 321b, 322
network-249: 45c, 46a, 46a, 46c, 46d, 47b, 47c, 50c, 144
newarp6(): 334b, 336, 338
newipaux(): 40a, 41c, 45a, 123b
newIPICMP(): 472b, 474b, 475, 476, 477a, 477b

NEXT: 255b, 288, 289a, 290b
Nfs-244: 29f, 29g, 40a, 40b
Ngates-114: 384a
Nhash: 232a
NHASH-100: 126a, 130d, 374
NHASH-115: 106b, 333a
nhgetl(): 229a
nhgets(): 229b
nhgetv(): 228d
Nifstatqid: 240a, 242, 245, 256d, 318e
Nipht: 135e, 136c
Nipstats: 31a, 77b
NLHT-327: 406, 406, 412a, 432
Nlog-253: 189a, 190a, 389a
Nmaxaddr: 35f, 240a, 240e, 246c
Nmhash: 181d, 240a, 252b
NMiiPhy: 310, 314b, 316c
NMiiPhyr: 281, 314b
Nmtuqid: 240a
NNH: 498d
NOOPOPT-293: 406, 422, 424, 425, 426
NOW: 92b, 100c, 157, 160, 162b, 165, 166a, 234a, 334b, 336, 337b, 338, 340c, 360c, 361, 377c, 378a, 397b, 398a, 427, 430, 432, 437e, 438, 441, 449, 456, 462d, 492b, 512c, 517g
Nqt-74: 153, 162b, 392a, 401a
Nrb-784: 278c, 292
Nrd-782: 278c, 292
Nrtu-721: 276a
Nself-99: 374
Nstatistics-550: 271b, 278d, 280c, 281
Nstatqid: 240a, 242, 245, 256d, 318e
Nstats-212: 344b, 344c, 344d, 351a
Nstats-359: 410c, 411, 412a, 462c
Nstats-90: 154c, 154d, 168a
Nstats6-146: 468, 470f, 471a, 483
Nstformat-108: 378c
Ntd-783: 278c, 292
ntohtcp4(): 426, 441
ntohtcp6(): 425, 441
Ntypeqid: 240a, 242, 245
Ntypes: 255a, 257b, 261c, 319d, 323d
nullbind(): 216a, 216b
nullbwrite(): 216a, 216d
nullmedium: 216a, 216e
nullmediumlink(): 216e, 327
nullunbind(): 216a, 216c
Oflag-148: 469b, 481b
Ofst-453: 268a
openfile(): 244b, 250
optexsts: 478, 498b

Opthdr (typedef): 500
OptlenErrs6-143: 468, 478
OSPF: 498d
OutDiscards: 31a, 95c, 97b, 103b, 104e, 486e, 490
OutMsgs-208: 344b, 345e
OutMsgs-81: 154d, 157
OutMsgs6-136: 468
OutNoRoutes: 31a, 97c, 486e
OutOfOrder-348: 410c, 458c
OutOfOrder-85: 154d, 395b
OutRequests: 31a, 93, 486e
OutRsts-343: 410c, 428
OutSegs-338: 410c, 449
overflow(): 205b, 209
Ovw-430: 201c, 205b, 209, 267b
Owc-460: 268b
PacketTooBigV6-153: 469c, 477b, 481b
Page0-423: 199b, 200a, 201a, 203, 204, 207, 208b, 209, 267a
Page1-424: 200a, 200b, 208a, 211c, 267a
Page2-425: 267a
Par0-408: 200b, 266b
ParamProblemV6-156: 469c
parseaddr(): 246c, 252d
parsecmd(): 55a, 123b, 191d, 259, 282c, 321b
parseether(): 85a
parseip(): 59, 60, 72, 74, 82b, 89, 109b, 123b, 180c, 180d, 191d, 224d, 512c
parseipmask(): 72, 74, 82b, 123b, 224d
parsemac(): 77c, 109b, 372f
PassiveOpens-334: 410c, 421
Pba-622: 274b
Pbe-719: 276a
Pci66-587: 273a
Pcix100-593: 273a
Pcix133-594: 273a
Pcix66-592: 273a
Pcixmode-589: 273a
PcixspeedMASK-590: 273a
PcixspeedSHIFT-591: 273a
PcssMASK-731: 276c
PcssSHIFT-732: 276c, 289b
Phy-472: 268d
Phyidr1: 310, 314b
Phyidr2: 310, 314b
Pif-745: 277d
pktbind(): 367, 368a
pktbwrite(): 367, 368c
pktin(): 367, 368d
pktmedium: 367, 369a
pktmediumlink(): 327, 369a

pktunbind(): 367, 368b
Pmcf-708: 275d
POLY-797: 265a, 265b
POLY-800: 325b, 325c
prefixvals: 186, 226d
PREFIX_INFO: 498e
PREV: 255c, 286
PriMASK-778: 278b
printroute(): 123b, 125b
Prior-557: 272
PriSHIFT-779: 278b
Pro-465: 201c, 211c, 268c
procopts(): 493a, 494a
procsyn(): 441, 457b
procxtns(): 490, 493a
promiscuous(): 199b, 212a
Proto: 234b, 234b
PROTO-247: 35c, 46d, 47b, 47c, 48a, 49c, 50b, 52a, 55a, 61d, 61e, 62d, 62g, 64b, 64g, 65c, 65e, 193b, 193c, 193d, 328c, 329b, 330b
Proto (typedef): 234b
Prx-426: 201c, 205b, 267b
Prxok-467: 207, 268d
Ps0-421: 200b, 208a, 211c, 267a, 267a
Ps1-422: 200b, 208a, 211c, 267a, 267a
PSH-288: 406, 449, 453
Psp-713: 276a, 286
Pstart-395: 200a, 266b
Pstop-396: 200a, 266b
ptclbsum(): 230d
ptclcsum(): 90d, 140b, 145b, 157, 158, 160, 345e, 346a, 346b, 348a, 348c, 394, 397a, 422, 424, 441, 472a, 478, 514a, 515c
PthreshMASK-722: 276b
PthreshSHIFT-723: 276b, 286
Ptx-427: 201c, 205b, 209, 267b
Ptxok-454: 268b
PtypeIP-759: 278a
PtypeTCP-757: 278a
Qarp-217: 34i, 46b, 108a, 108b, 108c, 109a
Qclone-224: 34i, 47a, 47b, 52a
Qconvbase-227: 34i, 34i, 47c
Qconvdir-226: 34i, 46d, 47c, 49b, 51b
Qctl-228: 34i, 47d, 48b, 49c, 50b, 52a, 54d, 55a, 62g, 328c
Qdata-229: 34i, 47d, 48b, 49c, 50b, 61d, 61e, 328c, 329b, 330b
Qerr-230: 47d, 49c, 50b, 62b, 62c, 62d
QID-248: 35b, 40a, 45c, 46c, 46d, 47b, 47c, 48a, 52a, 62g
Qiproute-218: 34i, 46b, 108b, 120a, 120b, 120c
Qipselftab-219: 46b, 49b, 130a, 130b, 130c
Qlisten-231: 47d, 62e, 62f, 62g
Qlocal-233: 47d, 49b, 64h, 65a, 65c

Qlog-221: 46b, 189d, 189e, 189f, 189g, 189h, 189i
QMAX-102: 68d, 374
QMAX-268: 406, 417c, 420b, 430, 463b
Qndb-220: 46b, 181g, 181h, 181j, 182a, 182b, 182c
Qprotobase-223: 34i, 34i, 46d
Qprotodir-222: 34i, 45c, 46d, 47c, 49b, 51b
Qremote-234: 47d, 49b, 64h, 65a, 65e
Qsnoop-235: 47d, 192a, 193a, 193b, 193c, 193d
Qstats-225: 47a, 47b, 49b, 63c, 64b
Qstatus-232: 47d, 49b, 64d, 64e, 64g
Qtopbase-216: 34i, 34i, 45c
Qtopdir-215: 34i, 40a, 45c, 46d, 49b, 51b
QueryTime-63: 162b, 166a, 391b, 391b
Radv-529: 271b, 289b
Rah-554: 271b, 302b
Ral-553: 271b, 302b
rangecompare(): 116c, 117, 131b
Rbcast: 38e, 122a, 178a, 178b, 179a, 365b, 486e
Rbcr0-402: 199b, 201a, 204, 208b, 209, 266b
Rbcr1-403: 199b, 201a, 204, 208b, 209, 266b
Rbsz-785: 278c, 284b, 292
Rcontained-130: 116b, 116c, 117, 131b, 518e
Rcontains-129: 116b, 116c, 117, 131b, 518e
Rcr-404: 199b, 201c, 211c, 266b
Rctl-507: 271b, 283a, 289b, 290b, 301
Rd: 277c, 277c
Rd.addr: 277c
Rd.checksum: 277c
Rd.errors: 277c
Rd.length: 277c
Rd.special: 277c
Rd.status: 277c
Rd0-414: 267a, 267a
Rd1-415: 267a, 267a
Rd2-416: 267a, 267a
Rd (typedef): 277c
RdABORT-420: 199b, 200a, 201a, 203, 204, 207, 208b, 209, 267a
Rdbah-523: 271b, 289b
Rdbal-522: 271b, 289b
Rdc-432: 204, 208b, 267b
Rdd-739: 277d, 290b
Rdfh-517: 271b
Rdfhs-519: 271b
Rdfpc-521: 271b
Rdft-518: 271b
Rdfts-520: 271b
Rdh-525: 271b, 289b
Rdlen-524: 271b, 289b
rdread(): 208b, 265d

RdREAD-417: 204, 208b, 267a
RdSEND-419: 267a
Rdt-526: 271b, 289a, 289b
RdtmsEIGHTH-691: 275d
RdtmsHALF-689: 275d, 289b
RdtmsMASK-688: 275d
RdtmsQUARTER-690: 275d
Rdtr-527: 271b, 282c, 289b
rdwrite(): 204, 205a
RdWRITE-418: 204, 267a
readipifc(): 73c, 76c
readstr(): 54d, 64b, 64g, 65c, 65e, 182b, 245, 281
ReasmFails: 31a, 100c
ReasmOKs: 31a, 100c
ReasmReqds: 31a, 100c
ReasmTimeout: 31a, 100c
receive(): 205b, 207, 209
Recovery-353: 410c, 438
RecoveryCwind-357: 410c, 438
RecoveryDone-354: 410c, 438
RecoveryNoSeq-356: 410c, 438
RecoveryPA-358: 410c, 438
RecoveryRTO-355: 410c, 456
recvarp(): 363, 364
recvarpproc(): 77c, 364
Redirect-157: 469c
Redirect-191: 343a
RedirectV6-173: 469c, 478
REDIR_HEADER: 498e
Ref0-392: 205b, 266b
Ref1-393: 205b, 266b
Ref2-394: 201c, 205b, 266b
regr: 200b, 201a, 201c, 202b, 204, 205b, 208a, 208b, 209, 211c
regw: 199b, 200a, 200b, 201a, 201c, 202a, 203, 204, 205b, 207, 208a, 208b, 209, 211c
remselfcache(): 128b, 382b
Ren-678: 275d, 290b
Reop-740: 277d, 290b
Requals-128: 116b, 116c, 117, 131b, 518e
Reseq: 409a, 464
Reseq.bp: 409a
Reseq.length: 409a
Reseq.next: 409a
Reseq.seg: 409a
Reseq (typedef): 464
ReseqBytelim-349: 410c, 458c
ReseqPktlim-350: 410c, 458c
Resequenced-347: 410c, 458c
Reset-377: 197b, 270a
resolveaddr6(): 358, 361

RETRAN-310: 406, 438, 455c
Retrans-86: 154d
RetransSegs-339: 410c, 455c
RetransSegsSent-340: 410c, 449
RetransTimeouts-341: 410c, 456
ReTransTimer: 107c, 334a, 334a, 334b, 340c, 361
RETRANS_TIMER: 334a, 498e
reverse-476: 210b, 210c
Rfce-574: 272, 285b
Rflag-150: 469b
Rfollows-127: 116b, 116c, 117, 131b, 518e
rformat-133: 121a, 121a, 121b, 125b
RH: 493b, 498d
Rifc: 38e, 71c, 94c, 113d, 117, 122a, 132a, 381b, 486e, 508b
ringinit(): 199b, 200a, 207
Rmulti: 38e, 122a, 127b, 128b, 178a, 178b, 382a, 383c, 486e, 517f
Route: 234b, 234b
Route (typedef): 234b
routecmd-258: 191c, 191d
routeflush(): 123b, 132a, 132a
routelock-123: 114, 115a, 122b, 123b, 131a, 507c, 508a
RouterAddrErrs6-145: 468, 478
RouterAdvert-170: 469c, 478, 481b
routeread(): 120b, 120d
Routerparams: 234b
Routerparams (typedef): 234b
RouterSolicit-169: 469c, 478, 481b
RouteTree: 234b, 234b
RouteTree (typedef): 234b
routetype(): 121c, 122a, 130d
Routewalk: 234b, 234b
Routewalk (typedef): 234b
routewrite(): 120c, 123b
route_type: 38e
Routinghdr (typedef): 500
Rpreceeds-126: 116b, 116c, 117, 131b, 518e
Rproxy: 38e, 182e, 182f, 381b
Rps-763: 278a
Rptpt: 38e, 122a, 182d, 182e
rr(): 122b, 123a, 123a
Rrst-677: 275d
Rs-762: 278a, 288
Rsar0-400: 204, 208b, 266b
Rsar1-401: 204, 208b, 266b
Rsr-391: 266b
RST-289: 406, 428, 429, 441
Rst-433: 201a, 267b
RSVP: 498d
Rtlc-720: 276a

Runi: 38e, 71c, 94c, 122a, 125e, 148b, 149a, 178a, 346b, 365b, 486e
Rv4: 38e, 114, 115c, 116a, 116c, 118a, 121c, 122a, 123a, 131a, 132b, 379
Rxcfg-648: 274d
Rxchange-673: 275c
Rxconfig-674: 275c
Rxcsum-551: 271b, 289b
Rxcw-510: 271b
Rxdctl-528: 271b, 289b
Rxdmt0-644: 274d, 290b, 293
Rxe-428: 201c, 205b, 267b
Rxe-752: 277e
Rxinvalid-672: 275c
rxmitproc(): 107b, 107c
rxmitsols(): 107c, 340c
Rxnocarrier-671: 275c
Rxo-645: 274d, 290b, 293
rxready(): 107c, 341a
Rxseq-643: 274d, 290b, 293
Rxsync-675: 275c
Rxt0-646: 274d, 290b, 293
Rxword-670: 275c
Sbp-679: 275d
scalediv-92: 151b, 151c, 392e
scalednconv(): 150, 151d, 170
scalemul-93: 151b, 151c, 392e
Se-747: 277e
Seconds-59: 391b, 391b
Secrc-710: 275d
sendarp(): 358, 360c
sendgarp(): 27f, 356c, 362
Sep-461: 268c
Seq-748: 277e
seq_ge(): 437d, 438, 441
seq_gt(): 437c, 438, 441, 449
seq_le(): 437b, 438
seq_lt(): 415b, 437a, 438, 458c
seq_within(): 436, 441, 457a, 459b
setbit(): 210c, 211b
setfilter(): 210c, 211c, 212a
setladdr(): 56b, 59, 329c
setladdrport(): 56b, 58b, 59, 61c
setlport(): 56b, 58f, 59
setluniqueport(): 59, 329d
setraddrport(): 56b, 60
set_cksum(): 472a, 472d, 474a, 474b, 475, 476, 477a, 477b
sfixedformat: 375a, 375a
Sflag-149: 469b, 478, 481b
Shiftconv-240: 35a
Shiftproto-243: 35a

shutdown(): 199b, 201b
Sid-624: 274b
Sk-601: 273c, 298
slineformat: 375b, 375b
Slu-560: 272, 296b
sndrst(): 428, 441
sndsynack(): 430, 431b, 432
SOLN_PREF_LEN: 498e
Spdbyps-619: 274a
Spi-610: 273c, 300
sprintroute(): 120d, 121b
SrcQuench-155: 469c
SrcQuench-190: 343a
SRC_LLADDR: 474b, 478, 498e
SRC_UNI: 498e, 512b
SRC_UNSPEC: 340c, 474b, 498e, 512b, 517d
Sspeed10-564: 272
Sspeed100-565: 272, 285b
Sspeed1000-566: 272, 285b
SspeedMASK-562: 272, 285b
SspeedSHIFT-563: 272
ST: 498d
Sta-412: 199b, 203, 204, 207, 208b, 209, 267a
Statelen-252: 64b, 64c, 64g, 65c, 65e
Statistics-545: 271b, 281
statistics-792: 280c, 281
statnames-213: 344c, 351a
statnames-265: 31b, 77b
statnames-360: 411, 462c
statnames-91: 154e, 168a
statnames6-178: 471a, 483
Status-496: 271b, 296b, 302b
stformat-107: 130d, 378b, 378b
Stp-411: 200a, 201a, 207, 209, 267a
Svid-625: 274b
SwdpinshiMASK-612: 274a
SwdpinshiSHIFT-613: 274a, 275a
SwdpinsloMASK-569: 272
SwdpinsloSHIFT-570: 272, 275a
SwdpiohiMASK-614: 274a, 302b
SwdpiohiSHIFT-615: 274a, 275a, 302b
SwdpioloMASK-571: 272, 302b
SwdpioloSHIFT-572: 272, 275a, 302b
Swxoff-718: 276a
SYN-290: 406, 422, 424, 428, 430, 434b, 441, 449, 459b
SYNACK-312: 406, 434b, 438, 449
SYNACK_RXTIMER-306: 406, 432, 434b
Syn_received-318: 406, 414c, 415a, 441, 449, 462d
Syn_sent-317: 406, 412c, 414c, 415a, 419d, 421, 441, 449, 456, 461

Tadv-544: 271b, 286
TargetErrs6-142: 468, 478
targettype(): 481a, 481b
TARGET_LLADDR: 475, 498e
TARG_MULTI: 340c, 361, 498e, 517d
TARG_UNI: 474b, 498e
Tbcr0-398: 203, 266b
Tbcr1-399: 203, 266b
Tbimode-580: 273a, 296b, 302b
Tbt-513: 271b
Tcfi-777: 278b
TCP: 498d
Tcp: 408b, 464
Tcp.ack: 408b
Tcp.dest: 408b
Tcp.flags: 408b
Tcp.len: 408b
Tcp.mss: 408b
Tcp.seq: 408b
Tcp.source: 408b
Tcp.update: 408b
Tcp.urg: 408b
Tcp.wnd: 408b
Tcp.ws: 408b
Tcp4hdr: 407c, 464
Tcp4hdr.frag: 407c
Tcp4hdr.id: 407c
Tcp4hdr.length: 407c
Tcp4hdr.proto: 407c
Tcp4hdr.tcpack: 407c
Tcp4hdr.tcpcksum: 407c
Tcp4hdr.tcpdport: 407c
Tcp4hdr.tcpdst: 407c
Tcp4hdr.tcpflag: 407c
Tcp4hdr.tcplen: 407c
Tcp4hdr.tcpopt: 407c
Tcp4hdr.tcpseq: 407c
Tcp4hdr.tcpsport: 407c
Tcp4hdr.tcpsrc: 407c
Tcp4hdr.tcpurg: 407c
Tcp4hdr.tcpwin: 407c
Tcp4hdr.tos: 407c
Tcp4hdr.Unused: 407c
Tcp4hdr.vihl: 407c
Tcp4hdr (typedef): 464
TCP4_HDRSIZE-272: 406, 420a, 424, 426, 428, 430
TCP4_IPLEN-270: 406, 406, 424, 441
TCP4_PHDRSIZE-271: 406, 406, 424
TCP4_PKT-274: 406, 420a, 424, 426, 441

TCP4_TCBPHDRSZ-273: 406, 424
Tcp6hdr: 408a, 464
Tcp6hdr.ploadlen: 408a
Tcp6hdr.proto: 408a
Tcp6hdr.tcpack: 408a
Tcp6hdr.tcpcksum: 408a
Tcp6hdr.tcpdport: 408a
Tcp6hdr.tcpdst: 408a
Tcp6hdr.tcpflag: 408a
Tcp6hdr.tcpopt: 408a
Tcp6hdr.tcpseq: 408a
Tcp6hdr.tcpsport: 408a
Tcp6hdr.tcpsrc: 408a
Tcp6hdr.tcpurg: 408a
Tcp6hdr.tcpwin: 408a
Tcp6hdr.ttl: 408a
Tcp6hdr.vcf: 408a
Tcp6hdr (typedef): 464
TCP6_HDRSIZE-277: 406, 420a, 422, 425, 428, 430
TCP6_IPLEN-275: 406, 406, 422, 441
TCP6_PHDRSIZE-276: 406, 406, 422, 441
TCP6_PKT-279: 406, 420a, 422, 425, 441
TCP6_TCBPHDRSZ-278: 406, 422
Tcp (typedef): 464
tcpabcincr(): 417b, 438
tcpackproc(): 418, 421
tcpacktimer(): 416a, 417c, 420b
tcpadvise(): 170, 461
tcpannounce(): 170, 414b
tcpclose(): 170, 414c
tcpcongestion(): 416b, 438, 456
tcpconnect(): 170, 413a
tcpcreate(): 170, 417c
Tcpcs-743: 277d, 290b
Tcpctl: 409b, 464
tcpctl(): 170, 462b
Tcpctl.abcbytes: 409b
Tcpctl.acktimer: 409b
Tcpctl.backedoff: 409b
Tcpctl.backoff: 409b
Tcpctl.code: 409b
Tcpctl.cwind: 409b
Tcpctl.flags: 409b
Tcpctl.flgcnt: 409b
Tcpctl.irs: 409b
Tcpctl.iss: 409b
Tcpctl.kacounter: 409b
Tcpctl.katimer: 409b
Tcpctl.mdev: 409b

Tcpctl.mss: 409b
Tcpctl.nochecksum: 409b
Tcpctl.nreseq: 409b
Tcpctl.protohdr: 409b
Tcpctl.qscale: 409b
Tcpctl.rcv: 409b
Tcpctl.rerecv: 409b
Tcpctl.resent: 409b
Tcpctl.reseq: 409b
Tcpctl.reseqlen: 409b
Tcpctl.rttseq: 409b
Tcpctl.rtt_timer: 409b
Tcpctl.scale: 409b
Tcpctl.snd: 409b
Tcpctl.sndsyntime: 409b
Tcpctl.srtt: 409b
Tcpctl.ssthresh: 409b
Tcpctl.state: 409b
Tcpctl.time: 409b
Tcpctl.timer: 409b
Tcpctl.timeuna: 409b
Tcpctl.type: 409b
Tcpctl.window: 409b
Tcpctl (typedef): 464
Tcpe-750: 277e
tcpgc(): 170, 462d
tcpgo(): 419a, 438, 441, 449, 454b, 455a
tcphalt(): 419b, 419d, 437e, 438, 441, 449
tcphangup(): 429, 462b
tcpincoming(): 434b, 441
tcpinit(): 170
tcpinuse(): 170, 414a
tcpiput(): 170, 441
tcpkeepalive(): 420b, 454b
tcpkick(): 415a, 417c, 441
tcpmtu(): 420a, 427, 430
tcpoutput(): 414c, 415a, 416a, 421, 441, 449, 455c
tcpporthogdefense: 412b, 412b, 441, 453, 462a
tcpporthogdefensectl(): 462a, 462b
Tcppriv: 412a, 464
Tcppriv.ackprocstarted: 412a
Tcppriv.apl: 412a
Tcppriv.ht: 412a
Tcppriv.lht: 412a
Tcppriv.nlimbo: 412a
Tcppriv.stats: 412a
Tcppriv.timers: 412a
Tcppriv.tl: 412a
Tcppriv (typedef): 464

tcprcvwin(): 415b, 441, 449, 453, 463b
TCPREXMTTHRESH-307: 406, 449
tcprxmit(): 438, 455c, 456
tcpsendka(): 453, 454b
tcpsetchecksum(): 455b, 462b
tcpsetkacounter(): 441, 454a, 455a
tcpsetscale(): 420b, 434b, 441, 463b
tcpsetstate(): 412c, 414c, 419d, 421, 434b, 441
tcpsettimer(): 438, 456, 463a
tcpsndsyn(): 421, 427
tcpstart(): 413a, 414b, 421
tcpstartka(): 455a, 462b
tcpstate(): 170, 413b
tcpstates: 407a, 413b, 456
tcpstats(): 170, 462c
tcpsynackrtt(): 434b, 437e, 441
tcptimeout(): 420b, 456
Tcptimer: 407b, 464
Tcptimer.arg: 407b
Tcptimer.count: 407b
Tcptimer.func: 407b
Tcptimer.next: 407b
Tcptimer.prev: 407b
Tcptimer.readynext: 407b
Tcptimer.start: 407b
Tcptimer.state: 407b
Tcptimer (typedef): 464
TcptimerDONE-282: 406, 418
TcptimerOFF-280: 406, 419b, 434b
TcptimerON-281: 406, 417d, 418, 419a, 438, 441, 449
tcptrim(): 441, 459b
TCP_ACK-284: 406, 420b
TCP_CONNECT-305: 406, 413a, 421
TCP_LISTEN-304: 406, 414b, 420b, 421
tcp_irtt: 410b, 410b, 420b
Tcr-405: 199b, 201c, 209, 266b
Tctl-511: 271b, 285b, 286, 301
Td: 277f, 307b
Td.control: 277f
Td.status: 277f
Td (typedef): 307b
Tdbah-538: 271b, 286
Tdbal-537: 271b, 286
Tdd-767: 278b
Tdfh-532: 271b
Tdfhs-534: 271b
Tdfpc-536: 271b
Tdft-533: 271b
Tdfts-535: 271b

Tdh-540: 271b, 286, 288
Tdlen-539: 271b, 286
Tdt-541: 271b, 286, 288
Ten-712: 276a, 286
Teop-758: 278a, 288
Tfce-575: 272, 285b
Tfuture-95: 398b, 398c
Tidv-542: 271b, 286
tifc-104: 71c, 73a, 73a, 127b, 516g, 517f
TimeExceed-159: 469c
TimeExceed-193: 343a, 346a, 348c
TimeExceedV6-154: 469c, 477a, 481b
timerstate(): 417d, 418, 419a, 419b
Timestamp-161: 469c
Timestamp-195: 343a
TimestampReply-162: 469c
TimestampReply-196: 343a
Time_wait-325: 406, 441, 456
Tipg-512: 271b, 286
Torl-548: 271b, 281
tosctlmsg(): 81b, 81c
Totl-549: 271b, 281
Tpsr-397: 199b, 266b
transmit(): 199b, 202c
Trst-711: 276a
Tse-761: 278a
Tsr-385: 205b, 266b
ttlctlmsg(): 81e, 81f
Tu-770: 278b
Tuniproxy: 481a, 481b, 498e
Tunirany: 481a, 481b, 498e
Tunitent: 481a, 481b, 498e
Tuofl-734: 276c
Txcw-509: 271b, 302b
TxcwAne-669: 275b, 302b
TxcwAs-664: 275b, 296b, 302b
TxcwConfig-668: 275b
TxcwFd-659: 275b, 302b
TxcwHd-660: 275b
TxcwNpr-667: 275b
TxcwPauseMASK-661: 275b, 302b
TxcwPauseSHIFT-662: 275b, 275b, 302b
TxcwPs-663: 275b, 296b, 302b
TxcwRfiMASK-665: 275b
TxcwRfiSHIFT-666: 275b
Txdctl-543: 271b, 286
Txdmac-530: 271b, 286
Txdw-640: 274d, 288, 293
Txe-429: 201c, 205b, 209, 267b

Txoff-579: 273a
Txp-413: 200b, 203, 204, 208a, 208b, 209, 211c, 267a
Txqe-641: 274d
txstart(): 202c, 203, 205b
TYPE-245: 35e, 45b, 46b, 47a, 47d, 49a, 50a, 51a, 51c, 328c, 329b, 330b
typeinuse(): 246b, 246c
UDP: 498d
Udp4hdr: 138a, 405
Udp4hdr.frag: 138a
Udp4hdr.id: 138a
Udp4hdr.length: 138a
Udp4hdr.tos: 138a
Udp4hdr.udpcksum: 138a
Udp4hdr.udpdport: 138a
Udp4hdr.udpdst: 138a
Udp4hdr.udplen: 138a
Udp4hdr.udpplen: 138a
Udp4hdr.udpproto: 138a
Udp4hdr.udpsport: 138a
Udp4hdr.udpsrc: 138a
Udp4hdr.Unused: 138a
Udp4hdr.vihl: 138a
Udp4hdr (typedef): 405
UDP4_IPHDR_SZ-367: 138b, 140b, 146a
UDP4_PHDR_OFF-365: 138d, 140b, 145b
UDP4_PHDR_SZ-366: 138e, 140b, 145b
Udp6hdr: 405, 513b
Udp6hdr.hoplimit: 513b
Udp6hdr.len: 513b
Udp6hdr.nextheader: 513b
Udp6hdr.udpcksum: 513b
Udp6hdr.udpdport: 513b
Udp6hdr.udpdst: 513b
Udp6hdr.udplen: 513b
Udp6hdr.udpsport: 513b
Udp6hdr.udpsrc: 513b
Udp6hdr.viclfl: 513b
Udp6hdr (typedef): 405
UDP6_IPHDR_SZ-368: 514a, 514c, 515e
UDP6_PHDR_OFF-370: 514a, 514c, 515c
UDP6_PHDR_SZ-369: 514a, 514c, 515c
udpadvise(): 134, 404
udpannounce(): 134, 140a
Udpcb: 148c, 405
Udpcb.headers: 148d
Udpcb (typedef): 405
udpclose(): 134, 139b
udpconnect(): 134, 139c
udpcreate(): 134, 139a

udpctl(): 134, 148e
udpinit(): 134
udpiput(): 134, 142
udpkick(): 139a, 140b
Udpmaxxmit-375: 403b
Udppriv: 135c, 405
Udppriv.csumerr: 137b
Udppriv.ht: 135c
Udppriv.lenerr: 137b
Udppriv.ustats: 137b
Udppriv (typedef): 405
Udprxms-373: 403b
udpstate(): 134, 147c
Udpstats: 137c, 405
udpstats(): 134, 147b
Udpstats.udpInDatagrams: 137c
Udpstats.udpInErrors: 137c
Udpstats.udpNoPorts: 137c
Udpstats.udpOutDatagrams: 137c
Udpstats (typedef): 405
Udptickms-374: 403b
UDP_UDPHDR_SZ-364: 138c, 140b, 146a, 514a, 515e
UDP_USEAD7-372: 148f, 149a, 149c
unfraglen(): 486e, 493a, 493b
unknownv6-109: 379, 510b, 511a
Unreachable-189: 343a, 346b, 348c
UnreachableV6-152: 469c, 476, 481b
unreachcode-179: 471b, 481b
unreachcode-214: 348b, 348c
unspecifiedv6-110: 379, 510b, 511c
update(): 438, 441
Upe-680: 275d, 283a
URG-286: 406, 441, 459b
V4: 21g, 95a, 95c, 140b, 145b, 146a, 146b, 148b, 158, 336, 338, 358, 360a, 360b, 363, 365a, 365b, 379, 381c, 404, 420a, 420b, 428, 429, 430, 434b, 441, 449, 514b, 516a, 516c, 516f
v4addroute(): 71c, 114, 123b, 127b
v4delroute(): 123b, 128b, 131a, 132b
v4freelist-121: 115b, 115c, 116a
V4H-131: 112b, 113a, 114, 123a, 131a
v4lookup(): 93, 104e, 109b, 112a, 125a, 178a, 338, 508b
v4prefix: 20e, 21b, 113d, 121c, 123b, 127b, 128b, 186, 224d, 379, 508b, 514b, 516a, 516f
V4route: 234b, 234b
V4route (typedef): 234b
v4routegeneration-122: 37d, 112a, 113b, 114, 131a
v4tov6(): 21c, 98, 113d, 145b, 148b, 158, 336, 338, 346b, 347c, 350, 363, 379, 400b, 404, 441, 461
V6: 21g, 109b, 338, 358, 360a, 360b, 365a, 365b, 379, 381c, 383c, 404, 420a, 420b, 428, 429, 430, 434b, 441, 449, 481b, 486e, 514a, 514b, 515c, 515d, 515e, 516a, 516b, 516c, 516f
v6addrcurr-113: 379, 511b, 511c
v6addroute(): 123b, 127b, 507c, 516g, 517f

v6addrtype(): 379, 511a, 511c
v6allnodesL: 372a, 481b, 517f
v6allnodesLmask: 372b, 517f
v6allnodesN: 371e, 517f
v6allnodesNmask: 371f, 517f
v6delroute(): 123b, 128b, 132b, 508a
v6freelist-124: 507a, 518c, 518d
V6H-132: 507b, 507c, 508a, 508b, 515f
v6linklocal: 371c
v6linklocal-7: 501b, 504c
v6linklocalmask-8: 501b, 504d
v6llpreflen: 371d, 371d
v6llpreflen-9: 501b, 504e, 504e
v6lookup(): 109b, 125a, 338, 379, 381b, 486e, 490, 508b
v6loopback: 182d, 371b, 517f
v6loopback-6: 501b, 504b
v6mcpreflen-12: 501b, 505a, 505a
v6MINTU: 476, 477a, 477b, 498e
v6multicast-10: 501b, 504f
v6multicastmask-11: 501b, 504g
V6nd_9auth: 498e
V6nd_9fs: 498e
V6nd_home: 498e
V6nd_ip: 498e
V6nd_rdns: 498e
V6nd_srcaddrs: 498e
v6params: 234b, 234b
v6params (typedef): 234b
V6route: 234b, 234b
V6route (typedef): 234b
v6routegeneration-125: 507c, 508a, 508b, 519b
v6router: 234b
v6router (typedef): 234b
v6snpreflen-15: 501b, 505d, 505d
v6solicitednode: 372c, 372e
v6solicitednode-13: 501b, 505b
v6solicitednodemask-14: 501b, 505c
v6tov4(): 21d, 140b, 149b, 157, 160, 345e, 346a, 420b, 428, 430, 434b
v6Unspecified: 371a, 379, 474b, 478, 481b, 511c
valid(): 478, 481b
Vfe-704: 275d
Vid-627: 274b
VlanMASK-775: 278b
VlanSHIFT-776: 278b
Vle-765: 278a
Vme-576: 272
Vp-742: 277d
walkadd(): 118b, 118d, 118d, 131a, 507c, 508a
Wopenack-352: 410c, 449

WSOPT-296: 406, 422, 424, 425, 426
WS_LENGTH-297: 406, 422, 424, 425, 426
WthreshMASK-726: 276b, 286
WthreshSHIFT-727: 276b, 286, 289b
Wts-434: 199b, 267c
_dial_string_parse(): 353c, 354
_dp8390read(): 199a, 207, 208b
_freeifc(): 73c, 76a
_ilprocess(): 162a, 162b
_readipifc(): 73c, 74
__anon_enum_10: 392b
__anon_enum_11: 353a
__anon_enum_12: 372d
__anon_enum_13: 374
__anon_enum_14: 378c
__anon_enum_15: 510b
__anon_enum_16: 384a
__anon_enum_17: 333a
__anon_enum_18: 339
__anon_enum_19: 116b
__anon_enum_1: 226c
__anon_enum_20: 468
__anon_enum_21: 469a
__anon_enum_22: 469b
__anon_enum_23: 469c
__anon_enum_24: 357b
__anon_enum_25: 343a
__anon_enum_26: 343b
__anon_enum_27: 344a
__anon_enum_28: 344b
__anon_enum_29: 34i
__anon_enum_2: 233c
__anon_enum_30: 35a
__anon_enum_31: 329a
__anon_enum_32: 189a
__anon_enum_33: 191b
__anon_enum_34: 486a
__anon_enum_35: 217b
__anon_enum_36: 406
__anon_enum_40: 410c
__anon_enum_41: 417a
__anon_enum_42: 498d
__anon_enum_43: 498e
__anon_enum_44: 403b
__anon_enum_45: 270a
__anon_enum_47: 266b
__anon_enum_48: 267a
__anon_enum_49: 267b
__anon_enum_4: 240a

__anon_enum_50: 267c
__anon_enum_51: 268a
__anon_enum_52: 268b
__anon_enum_53: 268c
__anon_enum_54: 268d
__anon_enum_55: 314b
__anon_enum_56: 315a
__anon_enum_57: 315b
__anon_enum_58: 315c
__anon_enum_59: 315d
__anon_enum_5: 240f
__anon_enum_60: 316a
__anon_enum_61: 316b
__anon_enum_62: 271a
__anon_enum_63: 271b
__anon_enum_64: 272
__anon_enum_65: 273a
__anon_enum_66: 273b
__anon_enum_67: 273c
__anon_enum_68: 274a
__anon_enum_69: 274b
__anon_enum_6: 255a
__anon_enum_70: 274c
__anon_enum_71: 274d
__anon_enum_72: 275a
__anon_enum_73: 275b
__anon_enum_74: 275c
__anon_enum_75: 275d
__anon_enum_76: 276a
__anon_enum_77: 276b
__anon_enum_78: 276c
__anon_enum_79: 277a
__anon_enum_7: 152d
__anon_enum_80: 277b
__anon_enum_81: 277d
__anon_enum_82: 277e
__anon_enum_85: 278a
__anon_enum_86: 278b
__anon_enum_87: 278c
__anon_enum_88: 282a
__anon_enum_8: 391b
__anon_enum_9: 392a
__anon_struct_37.dupacks: 409b
__anon_struct_37.nxt: 409b
__anon_struct_37.partialack: 409b
__anon_struct_37.ptr: 409b
__anon_struct_37.recovery: 409b
__anon_struct_37.retransmit: 409b
__anon_struct_37.rto: 409b

__anon_struct_37.rxt: 409b
__anon_struct_37.scale: 409b
__anon_struct_37.una: 409b
__anon_struct_37.urg: 409b
__anon_struct_37.wl2: 409b
__anon_struct_37.wnd: 409b
__anon_struct_37: 409b, 409b
__anon_struct_38.ackptr: 409b
__anon_struct_38.blocked: 409b
__anon_struct_38.nxt: 409b
__anon_struct_38.scale: 409b
__anon_struct_38.urg: 409b
__anon_struct_38.wnd: 409b
__anon_struct_38.wptr: 409b
__anon_struct_38.wsnt: 409b
__anon_struct_38: 409b, 409b
__anon_struct_39.tcp4hdr: 409b
__anon_struct_39.tcp6hdr: 409b
__anon_struct_39: 409b, 409b
__anon_struct_4.v4: 37e
__anon_struct_4.v6: 516d
__anon_struct_46.id: 213a
__anon_struct_46.name: 213a
__anon_struct_46: 213a, 213a
__anon_struct_4: 37e
__anon_struct_83.addr: 277f
__anon_struct_83: 277f, 277f
__anon_struct_84.ipcse: 277f
__anon_struct_84.ipcso: 277f
__anon_struct_84.ipcss: 277f
__anon_struct_84.tucse: 277f
__anon_struct_84.tucso: 277f
__anon_struct_84.tucss: 277f
__anon_struct_84: 277f, 277f
__anon_struct_89.reset: 260b
__anon_struct_89.type: 260b
__anon_struct_89: 260b, 260b
__anon_struct_90.reset: 323a
__anon_struct_90.type: 323a
__anon_struct_90: 323a, 323a

Bibliography

[Com13] Douglas E. Comer. *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture*. Pearson, 6th edition, 2013. cited page(s) 13

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 13

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 13

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 13

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 13, 39, 184

[Pad16] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 23, 172, 174, 185

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994. cited page(s) 13

[Ste03] W. Richard Stevens. *UNIX Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley, 3rd edition, 2003. cited page(s) 13

[TW10] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Pearson, 5th edition, 2010. cited page(s) 13