TCP/IP Programming Interfaces Guide

CONCURRENT COMPUTER CORPORATION™

0890468-000
February 1998

The information for this guide was taken from the *Mentat TCP/IP Version 3.3; Volume 1, Programming Interfaces* manual with written permission from Mentant Inc.

Printed in U. S. A.

Revision History:		Level:	Effective With:
Original Release	-- February1998	000	PowerMAX OS Release 4.2

Contents

Chapter 1 TCP/IP Tunable Parameters

Named Dispatch 1-1
  ndd Program 1-1
  Setting and Retrieving Tunable Parameters 1-2
ARP Tunable Parameters 1-3
IP Tunable Parameters 1-3
  Notes on Path MTU Discovery 1-6
RAWIP Tunable Parameters 1-7
TCP Tunable Parameters 1-7
  Time Wait State 1-10
  Round Trip Time-out Interval 1-11
  Deferred ACKs 1-11
  Timer Thresholds 1-11
  Keep-alive Probes 1-12
  Forcing Connection Termination 1-12
UDP Tunable Parameters 1-13

Chapter 2 Transport Provider Interface

Introduction 2-1
TPI Primitives for TCP 2-1
  TCP State Machine 2-1
  T_ADDR_REQ and T_ADDR_ACK 2-3
  T_BIND_REQ and T_BIND_ACK 2-3
    Binding to Addresses and Ports Already in Use 2-4
    T_BIND_REQ “Address in use” Errors 2-5
      T_BIND_RE “Re-bind” Request Operation 2-5
  T_CONN_IND and T_CONN_RES 2-5
  Note on Connection Establishment 2-6
  T_CONN_REQ and T_CONN_CON 2-6
  T_[EX]DATA_IND and T_[EX]DATA_REQ 2-6
  T_DISCON_IND and T_DISCON_REQ 2-7
  T_INFO_REQ and T_INFO_ACK 2-7
  T_ORDREL_REQ and T_ORDREL_IND 2-8
  T_UNBIND_REQ 2-8
  Other TPI-Related Issues 2-8
  TCP Options 2-8
TPI Primitives for UDP 2-13
  T_ADDR_REQ and T_ADDR_ACK 2-13
  T_BIND_REQ and T_BIND_ACK 2-14
    Binding to Addresses and Ports Already in Use 2-14
    T_BIND_REQ “Address in use” Errors 2-15
    Broadcast Binds 2-15
  T_CONN_REQ and T_CONN_CON 2-16
  T_DISCON_REQ 2-16

T_INFO_REQ and T_INFO_ACK 2-16
T_UNBIND_REQ 2-17
T_UNITDATA_REQ and T_UNITDATA_IND 2-17
UDP Options 2-18
UDP Options for IP Multicast 2-22
TPI Primitives for RAWIP 2-23
T_BIND_REQ and T_BIND_ACK 2-24
Binding to a Protocol 2-24
Binding to an IP Address 2-24
T_CONN_REQ and T_CONN_CON 2-25
T_DISCON_REQ 2-25
T_INFO_REQ and T_INFO_ACK 2-26
T_UNBIND_REQ 2-26
T_UNITDATA_REQ and M_DATA 2-26
T_UNITDATA_IND 2-27
RAWIP Options 2-27
RAWIP Options for Multicast Routing 2-29
RAWIP Options for IP Multicast 2-31
Options Management 2-31
Semantics of T_OPTMGMT_REQ Message Formats 2-31
MGMT_flags Field in T_OPTMGMT_REQ 2-31
TLI Semantics for Options Processing 2-32
TLI Option Names 2-33

Chapter 3 IP Interfaces

Introduction 3-1
IP Module General Structure 3-1
ILLs and Interfaces 3-3
Loopback Device 3-4
Configuring an Interface 3-4
IP Configuration ioctl’s 3-5
Defining IP Routes 3-14
ARP ioctls 3-18

Ilustrations

Figure 2-1. TCP State Machine 2-2
Figure 2-2. Message Block Containing T_BIND_REQ Message Sent to IP 2-25
Figure 2-3. T_OPTMGMT_REQ with Three Options in TLI Format. 2-33
Figure 3-1. Configuration of IP as a Device Below TCP and UDP 3-2
Figure 3-2. Configuration of STREAMS Devices, ARP, and ILL 3-2
Figure 3-3. Relationship Between ILL’s and IPIF’s 3-3

Tables

Table 2-1. IP_REUSEADDR and IP_REUSEPORT for TCP 2-4
Table 2-2. TCP Values Returned. 2-7
Table 2-3. Binding to Addresses and Ports Already in Use 2-15
Table 2-4. UDP Returned Values 2-17
Table 2-5. Options Inserted Into T_UNITDATA_IND Messages. 2-18
Table 2-6. RAWIP Values 2-26
Table 2-7. MGMT_flags Field 2-31
Table 2-8. TLI Option Names. 2-33

Preface

Scope of Manual

This guide describes the tunable parameters and programming interfaces of TCP/IP for PowerMAX OS, based on Mentat TCP 3.3. This information is taken from the Mentat TCP V3.3 documentation, Volume 1; *Programming Interfaces*.

Structure of Manual

This guide consists of a title page, this preface, a master table of contents, three chapters, local tables of contents for the chapters, and an index.

- Chapter 1, *TCP/IP Tunable Parameters*, describes the TCP/IP tunable parameters for PowerMAX OS.
- Chapter 2, *Transport Provider Interface*, details the Transport Provider Interface (TPI) implementation used by TCP, UDP, and RAWIP modules in PowerMAX OS.
- Chapter 3, *IP Interfaces*, details the various interfaces to PowerMAX OS IP.

The index contains an alphabetical list of all paragraph formats, character formats, cross reference formats, table formats, and variables.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.
`list bold`	User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments

Referenced Publications

The following publications are referenced in this document:

A STREAMS-based Transport Provider Interface, Version 1.5 UNIX System Laboratories, 1992

X/Open Transport Interface (XTI) Documentation Number XO/CAE/91/600, X/Open Company Ltd., January 1992.

Internetworking with TCP/IP, Volume 1, D. E. Comer, Second Edition, Prentice Hall 1991.

Data Link Provider Interface, OSI Work Group, UNIX International, August 20, 1991.

1
TCP/IP Tunable Parameters

Named Dispatch 1-1
ndd Program 1-1
Setting and Retrieving Tunable Parameters 1-2
ARP Tunable Parameters 1-3
IP Tunable Parameters 1-3
Notes on Path MTU Discovery 1-6
RAWIP Tunable Parameters 1-7
TCP Tunable Parameters 1-7
Time Wait State 1-10
Round Trip Time-out Interval. 1-11
Deferred ACKs 1-11
Timer Thresholds 1-11
Keep-alive Probes 1-12
Forcing Connection Termination 1-12
UDP Tunable Parameters. 1-13

1
TCP/IP Tunable Parameters

Named Dispatch

Named Dispatch is an easy-to-use interface for direct communication between application programs and modules. Tunable parameters used by TCP/IP may be set and retrieved using this mechanism.

ndd Program

The **ndd** program provides a simple command line interface to the Named Dispatch facility. The program accepts arguments on the command line or may be run interactively. The general form of the command line is:

```
    ndd [-set | -get] device_name ndd_name [value]
```

where

-set	used to set a new value
-get	used to retrieve the currently stored value. If neither **-set** nor **-get** is supplied, **-get** is assumed
device_name	this is the name of the device to access, i.e. **/dev/tcp**, **/dev/ip** etc.
ndd_name	the name of the tunable parameter to modify, or to retrieve the current value of.
value	for **-set**, the new value of the tunable parameter. Numeric values may be decimal, octal (leading 0) or hexadecimal (leading 0x). A string must be included in quotes so that it is passed as a single argument to **ndd**.

If any of the arguments are missing, **ndd** will run in interactive mode, prompting for what's missing. If all arguments are omitted, **ndd** prompts for everything, as shown in the following (what you type is underlined; ↵ means type just the <ENTER> or <RETURN> key):

```
orion:/z/tmp> ndd -set /dev/tcp

name to get/set ? ?
?                              (read only)
tcp_time_wait_interval        (read and write)
tcp_conn_request_max           (read and write)
tcp_conn_grace_period          (read and write)
tcp_debug                      (read and write)
…
tcp_status                     (read only)
tcp_discon                     (write only)
tcp_discon_by_addr             (write only)
name to get/set ? tcp_debug
value ? ↵
length ? ↵
0
name to get/set ? tcp_debug
value ? 1
name to get/set ? ↵
module to query ? ↵
```

Use "?" to obtain a list of Named Dispatch variables for the current module. "**length**" is used only with **-get** to restrict the size of the ioctl buffer; this option is currently unused. Names marked "read only" work only with **-get**, producing a formatted report; these variables cannot be set. Names marked "write only" work only with **-set**, causing an action to be taken; they do not display any output.

Setting and Retrieving Tunable Parameters

The remaining sections of this chapter detail the various tunable parameters for each of the TCP/IP modules. Unless stated otherwise,

- Numbers are assumed to be in decimal; use "0x" prefix to specify hexadecimal values. Returned numbers are always displayed as decimal strings. For Boolean values, 0 is false, 1 is true.
- All times are specified in milliseconds, e.g., 240000 for 4 minutes.
- Default values and valid ranges are shown in brackets: "[minimum, maximum: default]".
- In general, all variables are global, i.e., they affect all instances of the module. Some settings take effect immediately, while others are used to initialize instance data and will only affect newly opened streams.

Some of the tunable parameters are not documented; undocumented parameters do not provide useful functionality in a deployed system.

ARP Tunable Parameters

arp_cache_report	Displays a report showing the ARP cache. [read-only]
arp_cleanup_interval	The amount of time that non-permanent, resolved entries are permitted to remain in ARP's cache. [`30000`, `3600000: 300000`]
arp_debug	Controls the level of debugging output from ARP module; 0 none, 1 minimal, 2 verbose. Meaningful only if ARP has been compiled with `ARP_DEBUG` defined. [`0`,`2:0`]

IP Tunable Parameters

ip_bogus_sap	To support DLPI Style 2 drivers, IP must temporarily bind to a nonstandard/ unused SAP during initial set-up. This parameter defines the SAP to use for this temporary bind operation. [`-,-:0x05fe`]
ip_check_subnet_addr	According to RFC 1122, Section 3.2.1.3, the subnet portion of a host address is not allowed to be all 0's or all 1's. Some implementations of TCP/IP do not enforce this. Therefore, enforcing this requirement on the address passed with the `SIOCSIFADDR` ioctl has been made optional. Set *ip_check_subnet_addr* to 1 to enforce the check; set to 0 to disable the check. [`0`,`1:1`]
ip_debug	Controls the level of debugging output from IP module; 0 none, 1 minimal, 2 verbose. Meaningful only if IP has been compiled with `IP_DEBUG` defined. [0,2:0]
ip_def_ttl	Sets the default time-to-live (TTL) in the IP header. [`1,255:255`]
ip_encap_ttl	Sets the time-to-live value inserted into the IP header which encapsulates a another IP packet. [`1,255:64`]
ip_encap_mc_only	IP packets with protocol `IPPROTO_IPIP` encapsulate another IP packet. Set *ip_encap_mc_only* to 1 to permit only multicast packets to be encapsulated in regular IP packets, or 0 to allow any IP packet to be encapsulated. If set to one, `IPPROTO_IPIP` packets containing non-multicast packets are dropped, not forwarded. [`0,1:1`]
ip_forward_directed_broadcasts	Set to 1 to have IP forward subnet broadcasts received on one interface out a different interface; set to 0 to inhibit forwarding. [`0,1:1`]

ip_forward_src_routed	Set to 1 to forward source routed packets; set to 0 to disable forwarding. If disabled, an ICMP Destination Unreachable message is sent to the sender of source routed packets needing to be forwarded. [`0,1:1`]
ip_forwarding	Controls how IP hosts forward packets: Set to 0 to inhibit forwarding (RFC 1122 compliance); set to 1 to forward always; set to 2 to forward only if the number of (logical) interfaces on the system is 2 or more (SunOS 4.1.x compatible). [`0,2:0`]
ip_fragment_timeout	RFC 1122 specifies 60 seconds as the time-out period for reassembly of IP datagrams. This is a long time, but may be appropriate for reassembly of datagrams that have traversed an internet. On local file server systems, on the other hand, fragmentation reassembly will either take place very quickly, or not at all, i.e., if all fragments are not received at about the same time, it is likely that one was dropped by the local interface, and will never arrive. In this case, keeping fragments around for 60 seconds may only exacerbate the problem. Therefore the time-out period is configurable using the *ip_fragment_timeout*; the actual value used is rounded to the nearest second. [`100,-:60000`]
ip_icmp_return_data_bytes	The maximum number of data bytes to return in ICMP error messages. [8,65536:64]
ip_ill_status	Display a report of all allocated instances of IP corresponding to a device (ILLs). [read-only] See Chapter 3, “IP Interfaces” for more information.
ip_ipif_status	Display a report of all allocated logical interfaces (IPIFs). [read-only] See Chapter 3, “IP Interfaces” for more information.
ip_ire_hash	Display a report of all routing table entries, in the order searched when resolving an address. [read-only]
ip_ire_status	Display a report of all routing table entries. Same information as *ip_ire_hash*, but format and ordering is different. [read-only]
ip_ire_cleanup_interval	This is the time-out interval for purging routing table entries. All entries unused for this period of time are deleted. [`5000,-:300000`]
ip_ire_flush_interval	All routing table entries are deleted after this amount of time, even those which have been recently used. [`60000,-:1200000`]
ip_ire_gw_probe_interval	This parameter controls the probe interval for Dead Gateway Detection. The implementation differs from that suggested in RFC 1122 and is based on periodic probing of active and dead gateways. *ip_ire_gw_probe_interval* con-

	trols the frequency of probing. With retries, the maximum time to detect a dead gateway is *ip_ire_gw_probe_interval* `+ 10000` milliseconds. Maximum time to detect that a dead gateway has come back to life is *ip_ire_gw_probe_interval*. [`15000,-:180000`]
ip_ire_pathmtu_interval	Every *ip_ire_pathmtu_interval* milliseconds, IP will scan its routing table for entries that have an MTU less than the MTU for the first hop interface. For each, it will increase the value to the next highest value in its internal table of common MTU sizes. In this way, if the path to a remote host has changed, and a larger MTU is now usable, the new MTU will be discovered. If this value is made too small, then excessive lost packets will result. [`5000,-:600000`]
ip_ire_redirect_interval	All routing table entries resulting from ICMP "Redirect" messages are deleted after this much time has elapsed, whether or not the entry has been recently used. [`60000,-:300000`]
ip_local_cksum	Set to 1 to force checksums for intramachine packets; 0 to bypass checksums on intramachine packets. Only meaningful if IP was compiled with `IP_DEBUG` defined. [`0,1:0`]
ip_loopback_bypass	Normally IP will loopback a packet for local addresses without sending the packet to the driver. Set this variable to 1 to force local packets to the driver (bypassing IP loopback logic). [0,1:0]
ip_max_bcast_ttl	This parameter controls the TTL inserted by IP into broadcast packets. IP expects upper layer protocols to fill in this field before sending the packet to IP. Setting *ip_max_bcast_ttl* to zero causes IP to accept this value. Otherwise, it restricts the value to be less than or equal to *ip_max_bcast_ttl*. [`0,255:1`]
ip_pmtu_strategy	Set the Path MTU Discovery strategy: 0 disables Path MTU Discovery; 1 enables Strategy 1; 2 enables Strategy 2 (see "Notes on Path MTU Discovery" on page 1-16). [`0,2:2`]
ip_reass_mem_limit	This parameter sets an upper bound on the number of bytes IP will use for packet reassembly. If the limit is reached, reassembly lists are purged until the space required for the new fragments becomes available. [`-,-:2000000`]
ip_respond_to_address_mask_broadcast	Set to 1 to make IP respond to incoming ICMP "Address Mask Request" packets received as broadcasts; set to 0 to ignore such requests. [`0,1:0`]

ip_respond_to_echo_broadcast	Set to 1 to make IP respond to incoming ICMP “Echo Request” packets received as a broadcast; set to 0 to disable. [`0,1:1`]
ip_respond_to_timestamp	Set to 1 to make IP respond to incoming ICMP “Time Stamp Request” packets; set to 0 to disable. [`0,1:1`]
ip_respond_to_timestamp_broadcast	Set to 1 to make IP respond to incoming broadcast ICMP “Time Stamp Request” packets; set to 0 to disable. [`0,1:1`]
ip_send_redirects	Set to 1 to have IP send ICMP “Redirect” packets; set to 0 to disable. [`0,1:1`]
ip_send_source_quench	Set to 1 to have IP send ICMP “Source Quench” packets when it encounters upstream flow control; set to 0 to disable. [`0,1:1`]
ip_strong_es_model	This option controls support for “Strong End System Model” described in RFC 1122, Section 3.3.4.2. When enabled, packet source addresses (and therefore interfaces on a multihomed host) affect selection of a gateway for outbound packets. Set to 0 to disable; set to 1 to enable. [`0,1:0`]

Notes on Path MTU Discovery

The current TCP/IP includes an implementation of “Path MTU Discovery” as described in RFC 1191. Because of problems encountered with some firewalls, hosts, and low-end routers, IP provides for selection of either of two discovery strategies, or for completely disabling the algorithm. The tunable parameter *ip_pmtu_strategy* determines the strategy.

Strategy 0:

This disables Path MTU discovery altogether.

Strategy 1:

All outbound datagrams have the “Don’t Fragment” bit set. This should result in notification from any intervening gateway that needs to forward a datagram down a path that would require additional fragmentation. When the ICMP “Fragmentation Needed” message is received, IP updates its MTU for the remote host. If the responding gateway implements the recommendations for gateways in RFC 1191, then the next hop MTU will be included in the “Fragmentation Needed” message, and IP will use it. If the gateway does not provide next hop information, then IP will reduce the MTU to the next lower value taken from a table of “popular” media MTUs

Strategy 2:

When a new routing table entry is created for a destination on a locally connected subnet, the “Don’t Fragment” bit is never turned on. When a new routing table entry for a non-local destination is created, the “Don’t Fragment” bit is not immediately turned on. Instead,

- An ICMP “Echo Request” of full MTU size is generated and sent out with the “Don’t Fragment” bit on.
- The datagram that initiated creation of the routing table entry is sent out immediately, without the “Don’t Fragment” bit. Traffic is not held up waiting for a response to the “Echo Request”.
- If no response to the “Echo Request” is received, the “Don’t Fragment” bit is never turned on for that route; IP won’t time-out or retry the ping. If an ICMP “Fragmentation Needed” message is received in response to the “Echo Request”, the Path MTU is reduced accordingly, and a new “Echo Request” is sent out using the updated Path MTU. This step repeats as needed
- If a response to the “Echo Request” is received, the “Don’t Fragment” bit is turned on for all further packets for the destination, and Path MTU discovery proceeds as for Strategy 1.

Assuming that all routers properly implement Path MTU Discovery, Strategy 1 is generally better—there is no extra overhead for the ICMP “Echo Request” and response. Strategy 2 is available only because some routers, or firewalls, or end hosts(!) have been observed simply to drop packets that have the DF bit on without issuing the “Fragmentation Needed” message. Strategy 2 is more conservative in that IP will never fail to communicate when using it.

RAWIP Tunable Parameters

rawip_bsd_compat	Set to 1 to adjust the length field in the IP header of inbound datagrams to exclude the length of the IP header itself; set to 0 to leave the length field as it exists within the IP datagram. [`0,1:1`]
rawip_def_ttl	Default Time to Live inserted into IP header[`1,255:255`]

TCP Tunable Parameters

tcp_conn_request_max	Maximum number of outstanding connection requests, e.g., an upper bound for `CONIND_number` in `T_BIND_REQ` messages. [`1,-:20`]
tcp_conn_grace_period	Additional time added to base time-out interval when sending a `SYN` packet. See “Round Trip Time-out Interval” on page 1-21. [`1,20000:500`]
tcp_debug	Set to 1 to enable global TCP debugging. This parameter is only meaningful if TCP has been compiled with the `TCP_DEBUG` flag defined. `tcp_debug` is not related to the `SO_DEBUG` option.[`0,1:0`]

tcp_deferred_ack_interval	Time-out interval for deferred ACKs. [1,60000:50]
tcp_discon	Terminate a TCP connection by specifying a TCP instance on the local machine. See "Forcing Connection Termination" on page 1-22. [write-only]
tcp_discon_by_addr	Terminate a TCP connection by specifying the local and remote addresses of the connection to terminate. See "Forcing Connection Termination" on page 1-22. [write-only]
tcp_dupack_fast_retransmit	TCP implements the fast retransmit logic suggested by RFC 1122, 4.2.2.21. Whenever TCP receives a series of ACKs without data, while unacknowledged data is present on the transmit list, TCP assumes that the remote TCP is requesting a retransmit. TCP temporarily reduces the congestion window to one MSS and retransmits. The number of ACKs needed to trigger a retransmit is *tcp_dupack_fast_retransmit*. [0,10:3]
tcp_ignore_path_mtu	Set this value to 1 to disable setting MSS from ICMP "Fragmentation Needed" messages. [0,1:0]
tcp_ip_abort_cinterval	Second timer threshold during connection establishment; see "Timer Thresholds" on page 1-21. [1000,-:240000]
tcp_ip_abort_interval	Second timer threshold for established connections; see "Timer Thresholds" on page 1-21. [500,-:240000]
tcp_ip_notify_cinterval	First timer threshold during connection establishment; see "Timer Thresholds" on page 1-21. [1000,-:10000]
tcp_ip_notify_interval	First timer threshold for established connections; see "Timer Thresholds" on page 1-21. [500,-:10000]
tcp_ip_ttl	TTL value inserted into IP header. [1,255:255]
tcp_keepalive_detached_interval	Interval for sending keep-alive probes when TCP is detached (see "Keep-alive Probes" on page 1-22). [10000,10*24*3600000:240000]
tcp_keepalive_interval	Interval for sending keep-alive probes (see "Keep-alive Probes" on page 1-22). [10000, 10*24*3600000: 2*3600000]
tcp_keepalives_kill	Type of keep-alive probe to use: set to 1 for "Keep-alive killer", 0 for "Keep-alive probe" (see "Keep-alive Probes" on page 1-22). [0,1:0]
tcp_largest_anon_ port	Largest anonymous port number to use. [1024,65535: 65535]
tcp_mss_def	MSS to use when none is available from IP or TCP MSS options. [1,65495:536]

tcp_mss_max	Upper limit on MSS. [`1,65495:65495`]
tcp_mss_min	Lower limit on MSS. [`1,65495:1`]
tcp_naglim_def	Initial value for the Nagle Limit for controlling when to send data, RFC 1122 4.2.3.4. The `TCP_NODELAY` option overrides this default on a per TCP basis, setting the Nagle Limit to 1. After initialization the Nagle Limit is computed from the current MSS. [`1,65535:65535`]
tcp_old_urp_interpretation	This parameter determines the byte to which the TCP urgent pointer points. Set to 0 for (correct) RFC 1122, Section 4.2.2.4 interpretation; set to 1 for (obsolete) RFC 793 interpretation. [`0,1:1`]
tcp_rexmit_interval_ initial	Initial value for round trip time-out, from which the retransmit time-out is computed (see “Round Trip Time-out Interval” on page 1-21). [`1,20000:500`]
tcp_rexmit_interval_max	Upper limit for computed round trip time-out (see “Round Trip Time-out Interval” on page 1-21). [`1,7200000: 60000`]
tcp_rexmit_interval_min	Lower limit for computed round trip time-out (see “Round Trip Time-out Interval” on page 1-21). [`1,7200000:200`]
tcp_smallest_anon_ port	First port number to use for anonymous bind requests. [`1024,65535: 49152`]
tcp_smallest_nonpriv_port	Smallest port number non-privileged processes may use. [`1024,32768:1024`]
tcp_snd_lowat_f raction	If nonzero, the TCP option `SO_SNDBUF` will set the transmit low water mark to `SO_SNDBUF` value divided by *tcp_snd_lowat_fraction*. [`0,16:0`]
tcp_sth_rcv_hiwat	If nonzero, sets the Stream head flow control high water mark. [`0,128000:0`]
tcp_sth_rcv_lowat	If nonzero, sets the Stream head flow control low water mark. [`0,128000:0`]
tcp_syn_rcvd_max	This parameter controls the `SYN` attack defense of TCP. The value specifies the maximum number of suspect connections that will be allowed to persist in `SYN_RCVD` state. For `SYN` attack defense to work, this number must be large enough so that a legitimate connection will not age out of the list before an `ACK` is received from the remote host. This number is a function of the speed at which bogus `SYN`s are being received and the maximum round trip time for a valid remote host. This is very difficult to estimate dynamically, but the default value of 500 has proven to be highly effective. [`1,10000;500`]

tcp_rcv_push_wait	In the absence of a `PSH` bit, a `FIN`, or a segment less than one MSS, the maximum number of bytes to accumulate before sending data upstream. [`0,128*1024:16384`]
tcp_rwin_credit_pct	The percentage of the current receive window that is sent upstream before **`canput`** is called. [`1,500:50`]
tcp_rwin_mss_ multiplier	If non zero, the initial receive window is this value times MSS; if zero, the parameter is not used, and the initial receive window is set to *tcp_recv_hiwater_def*. [`0,100:0`]
tcp_status	Use to obtain a complete **`netstat`**-like report on all TCP instances. [read-only]
tcp_text_in_resets	This variable controls whether or not TCP includes explanatory text with `RST` segments. RFC 1122 notes that a TCP SHOULD allow text in `RST` segments. When TCP generates a reset it logs the reason locally using **`strlog`**. By including text, the remote system can also log the cause of the `RST`. Since it is highly likely that some TCP implementation may not accept text in a `RST` segment, this feature may be turned off (for all TCP instances) by setting this variable to 0. [`0,1:1`]
tcp_time_wait_ interval	How long stream persists in `TCPS_TIME_WAIT` state. [`1000,600000:240000`]

Time Wait State

Figure 2-1 in Chapter 2 of this manual shows the state transition diagram for a TCP connection. When a connection is closed, it can go into `TIME_WAIT` state. The definition of TCP requires that a connection remains in this state for a period of time, to allow for the arrival of packets which can exist in the network before they are discarded. This time period is determined by *tcp_time_wait_interval*, which has a default value of 240000ms (4 minutes).

While a connection is in this state, a new connection between the same addresses (ports and IP addresses) may, under some circumstances, fail to be set up. For example, if repeated rcp commands are being performed between two systems, an rcp connection may not be established. While the connection is being attempted, the connection on the initiating system is in `SYN_SENT` state, and the connection on the receiving system is in `TIME_WAIT`. When the default value of *tcp_time_wait_interval* is used, the TCP connection will time out before the connection on the receiving system exits the `TIME_WAIT` state, and the rcp command will fail.

This can be avoided by reducing the value of *tcp_time_wait_interval*. This variable shouldn't be reduced to below 60000ms (1 minute). This will avoid the situation where TCP connections time out before a connection moves out of the `TIME_WAIT` state.

Round Trip Time-out Interval

Absent any other information, TCP initializes the round trip time-out interval for new connections to *tcp_rexmit_interval_initial*. However, in some cases a better estimate may be known if a routing table entry for the destination address already exists, in which case TCP will use this estimate rather than *tcp_rexmit_interval_initial*. In either case, it may be desirable to permit initial `SYN` packets to have a longer time-out interval; *tcp_conn_grace_period* is added to TCP's time-out value for `SYN` packets.

Round trip estimates are updated periodically and are forced to be between *tcp_rexmit_interval_min* and *tcp_rexmit_interval_max*.

Deferred ACKs

TCP follows an `ACK` strategy which is an extension of the "every other segment" approach described in RFC 1122, 4.2.3.2. TCP will defer sending `ACK`s until it has received `ack_cnt Š ack_cur_max` bytes. Initially `ack_cur_max == 1 MSS`. Each time TCP is able to successfully defer `ACK`ing for `ack_cur_max` bytes, i.e., no timer expired for unacknowledged data, TCP increases `ack_cur_max` by one `MSS`, up to an absolute maximum of `ack_abs_max`.

Whenever an `ACK` is generated because of a time-out, `ack_cur_max` is reset to 1 `MSS` and `rack_abs_max` is reduced by 1 `MSS`, down to a minimum of `2 x MSS`. `ack_abs_max` is reduced to keep TCP from degenerating into timer-based `ACK`s. At best (notably when the remote TCP is sending with the slow-start strategy), TCP is able to reduce significantly the number of acknowledgments required to keep a connection going full speed. At worst, `ack_abs_max` decreases until TCP reaches the "ACK every other packet" strategy suggested by RFC 1122, 4.2.3.2.

Timer Thresholds

When it must retransmit because a timer has expired, TCP first compares the total time it has waited against two thresholds, as described in RFC 1122, 4.2.3.5. If the first threshold is exceeded, TCP notifies IP that it is having trouble with the current connection and requests IP to delete the routing table entry for this destination. The assumption is that if no `ACK` has been received for an extended period of time, there may be network routing problems and IP should try to find a new route.

If it has waited longer than the second threshold, TCP terminates the connection and if the stream is still open, TCP sends a **`T_DISCON_IND`** upstream.

Separate threshold values are maintained for connection setup and for established connections. The following table summarizes the threshold variables which may be set for all TCP instances by changing the tunable parameter.

Tunable Parameter	Usage
tcp_ip_notify_interval	1st threshold established connections
tcp_ip_abort_interval	2nd threshold established connections
tcp_ip_notify_cinterval	1st threshold during connections
tcp_ip_abort_cinterval	2nd threshold during connections

Keep-alive Probes

TCP supports two types of keep-alives; they differ in how they treat the absence of an `ACK` to a keep-alive probe. The two approaches are:

`Keep-alive killer` This approach either re-sends a `FIN` if one has already been sent, or it sends a 1-byte message repeating the sequence number of the last byte sent and `ACK`'d. If the segment is not `ACK`'d by the remote TCP, the normal retransmission time-out will eventually exceed `tcp_second_timer_threshold`, and the connection will be terminated.

`Keep-alive probe` This approach sends a zero-length segment containing an `ACK` for the last byte seen. If the segment does not elicit an `RST` from the remote TCP, the timer is simply reset for sending another probe later; the connection persists.

`tcp_killer_keepalives` governs which type of keep-alive will be used: 0 for “Keep-alive probe”, 1 for “Keep-alive killer”.

The time-out interval for both approaches is `tcp_keepalive_interval`. If any activity has occurred on the connection or if there is any unacknowledged data when the time-out period expires, the timer is simply restarted. With either approach, if the remote system has crashed and rebooted, it will presumably know nothing about this connection, and it will issue an `RST` in response to the `ACK`. Receipt of the `RST` will terminate the connection.

The distinction between the two approaches is that “Keep-alive probe” will maintain the connection until and unless an `RST` is received, whereas “Keep-alive killer” can time-out and terminate the connection without actually receiving an `RST` from the remote TCP.

Forcing Connection Termination

Two tunable parameters, *tcp_discon* and *tcp_discon_by_addr* may be used to force termination of an established TCP connection.

tcp_discon takes a single argument, the address (as a hex string) of the TCP instance data for the TCP connection to terminate. This address is displayed by the **`tcp_status`** command.

tcp_discon_by_addr also takes a single argument which specifies the local and remote IP addresses and ports of the connection to terminate. *tcp_discon_by_addr* is provided for application programs, such as an SNMP agent, which know the endpoint addresses of the connection to terminate.

For example, SNMP sets the TCP group variable `tcpConnState` to `deleteTCB` to terminate a connection. To implement this using *tcp_discon_by_addr*, provide the 24-byte argument that defines the connection to terminate. For example, if MIB fields are defined as

MIB Field	Value
`tcpConnState`	`deleteTCB (12)`
`tcpConnLocalAddress`	`192.1.2.3`
`tcpConnLocalPort`	`1024`
`tcpConnRemAddress`	`192.4.5.6`
`tcpConnRemPort`	`2049`

then the 24-byte hex string is “`c00102030400c00405060801`”, corresponding to **`tcpConnLocalAddress`** (“`c0010203`” == `192.1.2.3`), **`tcpConnLocalPort`** (“`0400`” == `1024`), **`tcpConnRemAddress`** (“`c0040506`” == `192.4.5.6`), and **`tcpConnRemPort`** (“`0801`” = `2049`). Note that no “`0X`” prefix is used for the hex string.

UDP Tunable Parameters

udp_def_ttl	Default Time-to-Live inserted into IP header. [`1,255:255`]
udp_do_checksum	Set to 1 to compute datagram checksums; 0 to compute only IP header checksums. [`0,1:1`]
udp_largest_anon_ port	Largest port number to use for anonymous bind requests. [`1024,65535:65535`]
udp_smallest_anon_port	Smallest port number to use for anonymous bind requests. [`1024,65535:49152`]
udp_smallest_ nonpriv_port	Smallest port number for non-privileged users. [`1024,32768:1024`]
udp_pass_up_icmp	Set the default behavior for processing inbound ICMP messages; the **`UDP_RX_ICMP`** option sets this on a per-stream basis.

	RFC 1122 says that UDP must pass all ICMP errors up to the application. UDP implements this by sending the error upstream in a `T_UDERROR_IND`. Of course this breaks many applications which are not expecting to see these messages. Therefore, *udp_pass_up_icmp* controls whether or not ICMP errors are delivered upstream to the application; set to 0 for no ICMPs upstream, or 1 to send them up. Note that the default behavior for BSD implementations is always to send errors up on connected UDP sockets (streams) and never on unconnected UDP sockets. This is behavior is achieved by setting *udp_pass_up_icmp* to 2. `[0,2:1]`
udp_pass_up_ options	Set the default behavior for including options in `T_UNITDATA_IND` messages sent upstream; the **`UDP_RECVOPTS`** option sets this on a per-stream basis. Set to 0 ignore options; set to 1 to include them.
udp_wroff_extra	Number of additional bytes to reserve in `M_DATA` message blocks for inserting IP options and Link Layer header. `[16,256:32]`

2
Transport Provider Interface

Introduction 2-1
TPI Primitives for TCP 2-1
TCP State Machine. 2-1
T_ADDR_REQ and T_ADDR_ACK 2-3
T_BIND_REQ and T_BIND_ACK 2-3
Binding to Addresses and Ports Already in Use 2-4
T_BIND_REQ “Address in use” Errors 2-5
T_BIND_RE “Re-bind” Request Operation 2-5
T_CONN_IND and T_CONN_RES 2-5
Note on Connection Establishment 2-6
T_CONN_REQ and T_CONN_CON 2-6
T_[EX]DATA_IND and T_[EX]DATA_REQ 2-6
T_DISCON_IND and T_DISCON_REQ. 2-7
T_INFO_REQ and T_INFO_ACK 2-7
T_ORDREL_REQ and T_ORDREL_IND 2-8
T_UNBIND_REQ 2-8
Other TPI-Related Issues 2-8
TCP Options. 2-8
TPI Primitives for UDP 2-13
T_ADDR_REQ and T_ADDR_ACK 2-13
T_BIND_REQ and T_BIND_ACK 2-14
Binding to Addresses and Ports Already in Use 2-14
T_BIND_REQ “Address in use” Errors 2-15
Broadcast Binds. 2-15
T_CONN_REQ and T_CONN_CON 2-16
T_DISCON_REQ. 2-16
T_INFO_REQ and T_INFO_ACK 2-16
T_UNBIND_REQ 2-17
T_UNITDATA_REQ and T_UNITDATA_IND 2-17
UDP Options 2-18
UDP Options for IP Multicast 2-22
TPI Primitives for RAWIP. 2-23
T_BIND_REQ and T_BIND_ACK 2-24
Binding to a Protocol. 2-24
Binding to an IP Address 2-24
T_CONN_REQ and T_CONN_CON 2-25
T_DISCON_REQ. 2-25
T_INFO_REQ and T_INFO_ACK 2-26
T_UNBIND_REQ 2-26
T_UNITDATA_REQ and M_DATA. 2-26
T_UNITDATA_IND. 2-27
RAWIP Options 2-27
RAWIP Options for Multicast Routing 2-29
RAWIP Options for IP Multicast 2-31
Options Management. 2-31
Semantics of T_OPTMGMT_REQ Message Formats. 2-31
MGMT_flags Field in T_OPTMGMT_REQ 2-31

TLI Semantics for Options Processing 2-32
TLI Option Names 2-33

2
Transport Provider Interface

Introduction

This chapter details the Transport Provider Interface (TPI) implementation used by TCP, UDP, and RAWIP modules in PowerMAX OS. It is assumed that the reader is familiar with the TPI specification as described in the *A STREAMS-based Transport Provider Interface, Version 1.5*[1] (TPI document). For each of the three protocol modules, TCP, UDP, and RAWIP, implementation details are provided which are not covered in the above document, including comprehensive lists of options supported by each of the modules.

The TPI document describes the **`T_OPTMGMT_REQ`**, but it does not specify the format of options specified in this primitive. The PowerMAX OS protocol modules use the UNIX System V syntax and semantics for **`T_OPTMGMT_REQ`**. The final section of this chapter describes the **`T_OPTMGMT_REQ`** in detail, for all protocol modules.

TPI provides only part of the interface to PowerMAX OS TCP/IP modules; ioctls are used for controlling IP routing and configuration. (Refer to Chapter 3 of this manual for details.)

For each protocol, the supported primitives are presented in alphabetic order, with paired `REQ` and `ACK` primitives described together. Each section concludes with a detailed description of each option which may be used with that protocol module.

TPI Primitives for TCP

TCP State Machine

Throughout this chapter occasional reference to TCP state is made. The TCP state machine shown in Figure 2-1 is a modification of Figure 6 in RFC 793 and Figure 12.13 in the *Internetworking with TCP/IP, Volume 1*[2]. The circles are the states. Labels on the arrows are of the form “a/b”, where “a” is the event causing the transition, and “b” is output (if any) from TCP as part of the transition.

1. *A STREAMS-based Transport Provider Interface, Version 1.5* UNIX System Laboratories, 1992.
2. *Internetworking with TCP/IP, Volume 1*, Comer D. E., Second Edition, Prentice Hall 1991.

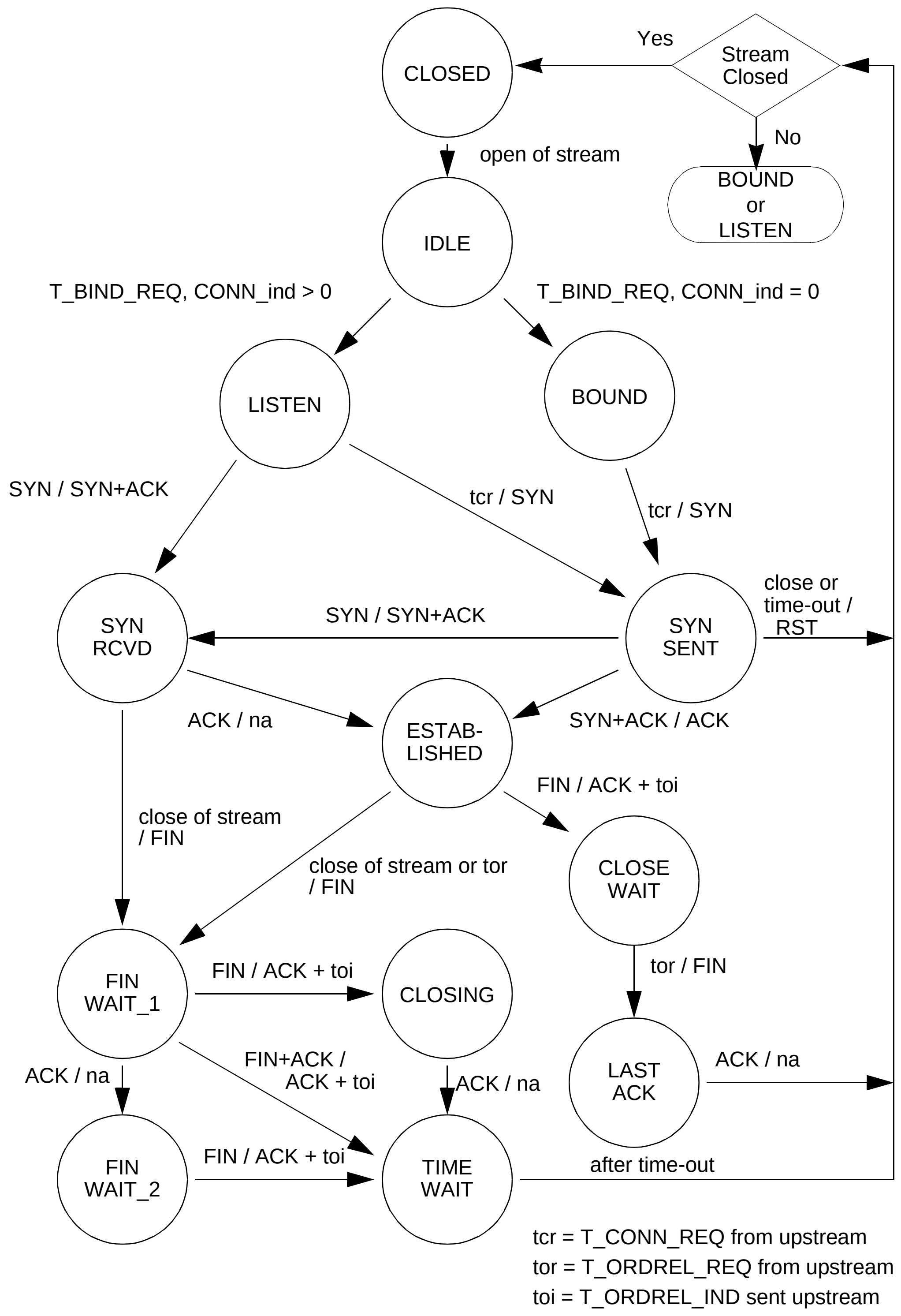

Figure 2-1. TCP State Machine

TCP states do not have a one-to-one correspondence to TPI states. In response to **`T_INFO_REQ`** messages from upper level modules and applications, TCP will return the TPI state which "best" corresponds to the current TCP state.

It is also important to note that TCP state transitions occur as the result of events detected by the TCP module, whereas state transitions for an upper layer interface such as TLI or Sockets occur as the result of events at the application level. Thus it is possible that the state reported by TCP will not be consistent with TLI or Socket state, the most extreme example being that under certain circumstances TCP maintains state after a stream is closed! It is the responsibility of the upper layer module or library to handle any such discrepancy correctly.

T_ADDR_REQ and T_ADDR_ACK

As indicated by *XPG4_1* flag in the *PROVIDER_flag* field of the **`T_INFO_ACK`**, TCP supports the **`T_ADDR_REQ`** primitive. The local address field in the **`T_ADDR_ACK`** is nonzero only if TCP state is `BOUND` or greater. The remote address field is nonzero whenever TCP state is greater than `SYN_SENT` as shown in Figure 2-1.

Note that if an endpoint in `BOUND` state has been bound to **`INADDR_ANY`** and any anonymous port, the IP address portion of the local address returned is zero; the length is *sizeof(struct sockaddr)*, which is 16. Once an endpoint is connected, the interface's actual IP address and port is returned, even if the endpoint had been bound to **`INADDR_ANY`**.

T_BIND_REQ and T_BIND_ACK

A **`T_BIND_REQ`** may be issued as the normal TPI `M_PROTO` message, or it may be specified as an *I_STR* ioctl with *ioc_cmd* field equal to **`TI_BIND`** (defined in **`timod.h`**), and the *ioc_dp* buffer containing the **`T_BIND_REQ`** request as it would appear in an `M_PROTO` message. The ioctl version allows TCP to check privilege for the request based on the current credentials of the endpoint, not those at the time the stream was opened. This may be relevant for applications which perform a "set uid" to bind to a privileged port (less than 1024) after the stream was opened.

If the *ADDR_length* field in the **`T_BIND_REQ`** is 0, the request is interpreted as a bind for **`INADDR_ANY`**.

If the *ADDR_length* field is nonzero, it must be 8 or 16, and the address specified at *ADDR_offset* must be a *sockaddr_in* structure. Even though "correct" applications will use an address length of 16, equal to *sizeof(struct sockaddr)*, a length of 8 is also supported for applications not including the zero pad in the length count as shown below:

```
struct sockaddr_in {
    short           sin_family;
    ushort          sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

The *sin_port* and *sin_addr* fields must be specified in Network Order (Big Endian); the *sin_family* field must be **AF_INET** or zero.

Binding to Addresses and Ports Already in Use

There are various situations which require a port to be reused. For example, server applications frequently need to open a new stream and bind it to the same address (port) that had been bound to a stream that has just been closed. Since the TCP state is not released immediately upon the close operation[3], the new bind request may fail with a TADDRBUSY error until the old TCP instance completes. TCP supports two options which prevent this error condition from occurring. **IP_REUSEADDR** and **IP_REUSEPORT** both permit the new bind operation to succeed, even if the old TCP instance has not yet completed; see *"Options Management" on page 2-57* for a description of options processing.

Table 2-1 below summarizes the conditions under which a **T_BIND_REQ** will succeed or fail, depending upon the setting of **IP_REUSEADDR** and **IP_REUSEPORT** and the specified IP address. In all cases, the **T_BIND_REQ** is specifying the same TCP port number as a port already in use by another TCP instance. **NewTCP** is the stream on which the current **T_BIND_REQ** is being sent; **OldTCP** is some stream previously bound to the same TCP port. IP1 and IP2 are two specific, distinct IP addresses; ANY is **INADDR_ANY (0.0.0.0)**. The rows across the top show eight different combinations of setting these options. RA stands for **IP_REUSEADDR**; RP stands for **IP_REUSEPORT** (the setting of **IP_REUSEADDR** for **OldTCP** does not matter). An X in a cell means the **T_BIND_REQ** will succeed; an empty cell means the bind will fail.

Table 2-1. IP_REUSEADDR and IP_REUSEPORT for TCP

RA for NewTCP RP for NewTCP RP for OldTCP		off off off	off off on	off on off	off on on	on off off	on off on	on on off	on on on
OldTCP	**NewTCP**								
IP1	IP1				X				X
IP1	IP2	X	X	X	X	X	X	X	X
IP1	ANY				X	X	X	X	X
ANY	IP1				X	X	X	X	X
ANY	ANY				X				X

In addition, note that:

- Only one stream receives packets (unlike UDP).
- **IP_REUSEPORT** is "cooperative", that is, both streams must have it set for it to apply. When set, all binds succeed.

3. The default behavior follows RFC 1122; tunable parameters may be used to override or minimize the time before state is released. See Chapter 1, "TCP/IP Tunable Parameters" for more information.

- Streams bound to **`INADDR_ANY`** receive packets only if there is no available "listening stream" with a specific IP address, i.e., a stream which issued a **`T_BIND_REQ`** with a specific IP address and a *CONIND_number* greater than zero and which currently has fewer than *CONIND_number* connection indications pending.
- If two streams are bound to the same explicit IP address, the stream receiving packets cannot be determined.

T_BIND_REQ "Address in use" Errors

Whenever the process of the previous paragraph results in an IP address/port number being unavailable, the TPI document dictates that the provider must return a `TADDRBUSY` error. Some early TLI libraries relied upon the provider returning a **`T_BIND_ACK`** with a different port than requested rather than a **`T_ERROR_ACK`**. In accordance with UNIX System V TLI convention, TCP will return a **`T_BIND_ACK`** with a different (unused) port than was specified in the original **`T_BIND_REQ`**.

T_BIND_RE "Re-bind" Request Operation

To support socket semantics, TCP supports an extension to TPI which permits an upper layer to change the *CONIND_number* for a bound stream using a second **`T_BIND_REQ`** without first issuing a **`T_UNBIND_REQ`**. This "re-bind" operation can be used only if the stream is already bound. To specify a new *CONIND_number*, issue a **`T_BIND_REQ`** with *ADDR_length == 0* and *CONIND_number* equal to the new value to replace the value from the previous bind.

Additionally, socket semantics allow an application to listen with a backlog of zero. This means that there is a listener for the port, but it is not accepting any connections. To effect this behavior, a TCP stream must first bind with a nonzero *CONIND_number*, and then re-bind with *CONIND_number == 0*. This action will cause TCP to issue a `RST` for any connection attempt, which in turn will cause the remote client to fail with `ECONNREFUSED`.

T_CONN_IND and T_CONN_RES

If multiple streams are bound with *CONIND_number* greater than zero (**listener streams**) to the same IP address and port number, the stream which receives a **`T_CONN_IND`** as the result of an inbound connection request is indeterminate. If a selected listener stream has more than *CONIND_number* unaccepted connect indications pending, TCP will attempt to locate another listener stream for the specified port.

The preferred mechanism for accepting a connection is to use the **`I_FDINSERT`** ioctl to insert the accepting stream's device read queue into the *QUEUE_ptr* field of the **`T_CONN_RES`**.

If the accepting stream is different than the listening stream, then the listening stream must be bound and the *CONIND_number* must be zero.

Note on Connection Establishment

As a result of normal processing of the TCP **SYN-ACK** handshake, TCP may already have created an established connection before the **T_CONN_IND** is seen by the application. This behavior is not arbitrary; certain well-known implementations of **ftp** rely upon this behavior. As a result, if a **T_CONN_IND** is neither accepted nor rejected, or equivalently, a **T_CONN_RES** or **T_DISCON_REQ** fails with a **T_ERROR_ACK** for a badly formed primitive, then the established connection will remain in force until the listening stream closes or the remote TCP issues a RST. It is the responsibility of the application to ensure that **every T_CONN_IND** is correctly acknowledged.

If the remote TCP that initiated the connection is a TPI provider, then that TCP will send a **T_CONN_CON** upstream when the **SYN-ACK** handshake completes. If the accepting TCP issues a **T_DISCON_REQ**, this causes a RST segment to be transmitted, which in turn causes the remote TCP to send a **T_DISCON_IND** upstream. Thus, if the accepting TCP fails to acknowledge a **T_CONN_IND**, the remote TCP mistakenly perceives that the connection was accepted and begins sending data. Since the local TCP has no stream to which to send the received data, it closes its receive window and the remote TCP hangs.

T_CONN_REQ and T_CONN_CON

TCP will allow a connection request to be issued on a stream immediately after an exchange of orderly releases, even though the local TCP instance may still be in **TIME_WAIT** state.

In general, streams on which a **T_CONN_REQ** will be sent should be bound with *CONIND_number* equal to zero. **T_CONN_REQ**s on a listening stream may fail if connection indications arrive for the stream.

A **T_CONN_REQ** may be issued on an unbound stream, and TCP will perform a bind to an anonymous port before processing the connection request.

Options may be supplied with the **T_CONN_REQ**. The format is an option buffer as would be supplied in a **T_OPTMGMT_REQ** with flag **T_NEGOTIATE** (refer to the *"Options Management" on page 2-57*). The options are set before the connection is started; the result is indistinguishable from having sent a separate **T_OPTMGMT_REQ** before the **T_CONN_REQ**.

The **T_CONN_CON** is sent upstream when the TCP **SYN-ACK** handshake has completed and the local TCP has transitioned to ESTABLISHED state. If provided with the **T_CONN_REQ**, the options buffer is returned on the **T_CONN_CON**.

T_[EX]DATA_IND and T_[EX]DATA_REQ

The *MORE_flag* is ignored in **T_[EX]DATA_REQ** primitives and always zero in **T_[EX]DATA_IND** primitives.

The last byte of data in a **T_EXDATA_REQ** is marked as TCP urgent data. Received urgent data is sent upstream in a **T_EXDATA_IND** containing the single urgent byte. An

M_PCSIG message with signal **SIGURG** is sent upstream when the first segment containing the urgent flag is received by TCP.

Note that TCP urgent data is not out-of-band and that T_EXDATA_IND M_PROTO messages are subject to flow control.

T_DISCON_IND and T_DISCON_REQ

T_DISCON_IND primitives are sent upstream when TCP receives a reset (RST flag set in a TCP segment). A **T_DISCON_IND** is sent up a listening stream to “cancel” a previous **T_CONN_IND** that has not yet been accepted; the *SEQ_number* field of the **T_DISCON_IND** will match that of the **T_CONN_IND** being cancelled.

In general, a **T_DISCON_REQ** is used to reject an inbound **T_CONN_IND**, and **T_ORDREL_REQ** is used to terminate an established connection. However, if a **T_DISCON_REQ** is sent to a TCP in **ESTABLISHED** state, a reset will be sent to the remote TCP.

As specified in the TPI document, if a TCP connection is in established state, or has initiated an orderly release, a **T_DISCON_IND** will be preceded by an M_FLUSH message to flush read and write queues. Likewise, an M_FLUSH is sent when a **T_DISCON_REQ** is received, and before the corresponding **T_OK_ACK** is returned.

If a stream had been in **ESTABLISHED** state, after processing a **T_DISCON_REQ** or **T_DISCON_IND** the stream is returned to its previous bound state.

T_INFO_REQ and T_INFO_ACK

Version 1.5 of the TPI documentation introduced a new *XPG4_1* flag in the *PROVIDER_flags* field of the **T_INFO_ACK** structure. This flag is set if it is defined in the current implementation. Table 2-2 shows the values returned by TCP:

Table 2-2. TCP Values Returned

Field	Return Value
TSDU_size	0
ETSDU_size	-1
CDATA_size	-2
DDATA_size	-2
ADDR_size	16
OPT_size	STRCTLSZ
TIDU_size	current MSS

Table 2-2. TCP Values Returned (Cont.)

Field	Return Value
SERV_type	`T_COTS_ORD`
CURRENT_state	TPI state of TCP
PROVIDER_flag	`XPG4_1`

T_ORDREL_REQ and T_ORDREL_IND

TCP will attempt an orderly release when a stream is closed.

If a **`T_ORDREL_REQ`** is sent on a **`T_IDLE`** stream, the message is discarded and no error is returned. No **`T_ERROR_ACK`** is generated for this out-of-state error condition, since TCP cannot distinguish it from a legitimate message that was sent while TCP was in **`ESTABLISHED`** state but which arrived in TCP after a reset was received from the remote TCP.

T_UNBIND_REQ

A **`T_UNBIND_REQ`** on a listening stream with outstanding connection indications will fail with **`TOUTSTATE`**. A **`T_UNBIND_REQ`** may immediately follow an exchange of **`T_ORDREL_REQ`** and **`T_ORDREL_IND`** messages, even though the underlying TCP instance may still be in **`TIME_WAIT`** or some other closing state (refer to Figure 2-1).

Other TPI-Related Issues

The Stream head uses the *q_maxpsz* and *q_minpsz* values of the top most module on a stream to determine how to process **`write`** system calls. If *q_minpsz* is zero, the Stream head will split large data buffers into multiple `M_DATA` messages of size *`q_maxpsz`*.

Since TCP overhead is least when processing message blocks that are a multiple of the current MSS, TCP changes the value of *`q_maxpsz`* of the top most module on the stream whenever MSS is set or changed. The initial default MSS is 536, which is changed once a connection is established to agree with the MSS for the virtual circuit.

TCP Options

TCP supports the following options, listed alphabetically by their XTI name. See *"Options Management" on page 2-57* for details on **`T_OPTMGMT_REQ`** formats; see *"TLI Option Names" on page 2-59* for a mapping of TLI (socket) option names to XTI name. "Inherited" below an option name means that an accepting stream inherits the current value from the listening stream; otherwise the value for the accepting stream is the system-wide default. Boolean values are true/on/yes (nonzero) and false/off/no (zero); by convention with sockets, nonzero values of some boolean values is the value of the option name.

`IP_BROADCAST` **level: `INET_IP`**
(Inherited) **len: sizeof(long)**

This option is simply recorded by TCP and IP. It has no effect.

Default: Off.

`IP_DONTROUTE` **level: `INET_IP`**
(Inherited) **len: sizeof(long)**

This option is simply recorded by TCP and passed through to IP. When value is set nonzero, the TTL in outbound IP packets is set to 1, causing packets not to be forwarded through a router.

Default: Off.

`IP_OPTIONS` **level: `INET_IP`**
len: —

For **`T_NEGOTIATE`**, the argument contains an IP header exactly as it would appear in an IP packet, e.g., data in network (Big Endian) byte order. **`len`** is rounded up to a multiple of 4 and the argument is padded with zeros, if necessary. The option is copied into the IP packet immediately after the IP simple header, i.e., starting at byte offset 20.

For **`T_CURRENT`**, if a source routing option is present, the current destination IP address is inserted as the first source route entry of the returned option.

Default: No options.

`IP_REUSEADDR` **level: `INET_IP`**
`IP_REUSEPORT` **len: sizeof(long)**
(Inherited)

See “T_BIND_REQ and T_BIND_ACK” on page 2-29 for a discussion of these options.

Default: Off.

`IP_TOS` **level: `INET_IP`**
len: sizeof(char)

The option value is inserted, without verification, into the TOS field of all subsequent IP packets.

Default: Zero.

`IP_TTL` **level: `INET_IP`**
len: sizeof(char)

The option value is inserted, without verification, into the TTL field of all subsequent IP packets.

Default: Tunable parameter *tcp_def_ttl* whose default value is 255.

`TCP_ABORT_THRESHOLD` **level: `INET_TCP`**
`TCP_NOTIFY_THRESHOLD` **len: sizeof(long)**

In any of several situations in which it must retransmit because a timer has expired, TCP first compares the total time it has waited against two thresholds, as described in RFC 1122, 4.2.3.5. These options set the corresponding threshold values for the current TCP stream. Tunable parameters exist to set the thresholds on a system-wide basis. See Chapter 1, "TCP/IP Tunable Parameters" for more information on these options and the corresponding tunable parameters.

The option value is the threshold time in milliseconds.

Default The default values for these options are the current values of the tunable parameters *tcp_ip_abort*_interval and *tcp_ip_notify_interval*, respectively.

`TCP_CONN_ABORT_THRESHOLD` **level: `INET_TCP`**
`TCP_CONN_NOTIFY_THRESHOLD` **len: sizeof(long**)

These options are the same as **`TCP_ABORT_THRESHOLD`** and **`TCP_NOTIFY_THRESHOLD`**, except that these values are used during connection establishment. See Chapter 1, "TCP/IP Tunable Parameters" for more information.

The option value is the threshold time in milliseconds.

Default: The default values for these options are the current values of the tunable parameters *tcp_ip_abort_cinterval* and *tcp_ip_notify_cinterval*, respectively.

`TCP_KEEPALIVE` **level: `INET_TCP`**
len: sizeof(long)

This option determines whether keep-alive packets are sent. If the value of the option is nonzero, keep-alive packets are sent; if zero, no keep-alive packets are sent. The actual value of the option is ignored; the keep-alive interval is the tunable parameter *cp_keepalive_interval.*

The two different keep-alive strategies are:

Keep-alive Probe	The probe is a zero-length segment containing an **`ACK`** for the last byte seen. If the probe is not **`ACK`**ed by the remote TCP, the timer is simply reset for sending another probe later; the connection persists.
"Killer" Keep-alives	The probe either re-sends a **`FIN`** if one has already been sent, or it sends a 1-byte message repeating the sequence number of the last byte sent (equal to last byte **`ACK`**ed here). If the probe is not **`ACK`**ed by the remote TCP, the normal retransmission time-out will eventually be exceeded, and the connection will be terminated.

Both approaches simply reset the timer if any activity has occurred on the connection or if there is any unacknowledged data. Likewise, if the remote system has crashed and rebooted, it will presumably know nothing about this connection, and it will issue a **RST** in response to the **ACK**. Receipt of the **RST** will terminate the connection.

The distinction between the two approaches is that "Keep-alive Probe" will maintain the connection until and unless a **RST** is received, whereas "Killer Keep-alive" can time-out and terminate the connection without actually receiving a **RST** from the remote TCP.

A tunable parameter, *tcp_killer_keepalives* governs which type of keep-alive will be used; the default value is 0, corresponding to using "Keep-alive Probe". See Chapter 1, "TCP/IP Tunable Parameters" for more information on TCP's tunable parameters.

Default: No keep-alive probes are sent.

TCP_MAXSEG **level: INET_TCP**
len: sizeof(long)

This "read only" option returns the current value of TCP's MSS.

TCP_NODELAY **level: INET_TCP**
len: sizeof(long)

Normally TCP accumulates small write requests before sending data, according to the Nagle Limit, RFC 1122 4.2.3.4. The **TCP_NODELAY** option overrides this default behavior on a per TCP basis, setting the Nagle Limit to 1.

Default: Off.

TCP_NOTIFY_THRESHOLD **level: INET_TCP**

See **TCP_ABORT_THRESHOLD**.

TCP_OOBINLINE **level: INET_TCP**
len: sizeof(long)

This option is simply recorded by TCP; it has no effect.

Note that "Out of band" data must be handled by an upper layer module or library; TCP will send urgent data upstream in a T_EXDATA_IND message (refer to the *T_[EX]DATA_IND and T_[EX]DATA_REQ* section in this chapter). These messages are always "in-line"; TCP itself has no concept of "expedited data" in the XTI sense, nor "Out of band" data in the Sockets sense.

Default: Off.

TCP_URGENT_PTR_TYPE **level: INET_TCP**
len: sizeof(long)

TCP urgent data pointer definition changed with RFC 1122. Set this option nonzero to use the old definition; set to zero for the current RFC 1122 definition.

Default: Zero.

XTI_DEBUG **level: XTI_GENERIC**
len: sizeof(long)

This option is simply recorded by TCP; it has no effect.

Default: Off.

XTI_LINGER — **level: XTI_GENERIC**
len: sizeof(struct linger)

Use **XTI_LINGER** to specify whether TCP performs an orderly release or an immediate disconnect (reset) when a stream closes. The argument is passed in a linger structure as shown below:

```
struct linger {
    int     l_onoff;
    int     l_linger;
}
```

Set *l_onoff* to zero to disable option; TCP will perform an orderly release on close. Set *l_onnoff* nonzero and *l_linger* to zero to force an immediate disconnect on close. A positive value for *l_linger* is equivalent to disabling the option; the value itself is ignored, i.e., the length of the "linger time" cannot be set.

Default: Disabled, i.e., TCP performs an orderly release on close.

XTI_PROTOTYPE — **level: XTI_GENERIC**
len: sizeof(long)

This option is provided for compatibility with other modules, notably RAWIP, that support multiple protocols; it has no effect in TCP. Only **IPPROTO_TCP** may be set in a **T_NEGOTIATE** request.

Default: **IPPROTO_TCP**.

XTI_RCVBUF — **level: XTI_GENERIC**
(Inherited) — **len: sizeof(long)**

The value in the option is used by TCP to compute its receive window size. The actual value set depends upon TCP state, the current receive window size, whether or not the remote TCP does window scaling, etc. In general, the value specified will be rounded up to a multiple of the MSS, and the result will be applied only if it doesn't shrink the current window or violate the value already "advertised" to the remote TCP.

You can set **XTI_RCVBUF** on the listener stream and that value will be transferred to the accepting stream as part of **T_CONN_RES** processing. You may also set **XTI_RCVBUF** after a connection is established to increase the window size.

Default: Tunable parameter *tcp_recv_hiwater_def*, whose default value is 32768.

TI_RCVLOWAT — **level: XTI_GENERIC**
(Inherited) — **len: sizeof(long)**

This option sets TCP's read-side queue *q_lowat* to the value given. In addition, it limits the amount of data (without the *PSH* or *FIN* bits set) that TCP will accumulate before sending data upstream.

Default: Tunable parameter *tcp_rcv_push_wait*, whose default value is 16384.

`XTI_SNDBUF`	**level: `XTI_GENERIC`**
(Inherited)	**len: sizeof(long)**

`XTI_SNDBUF` is used to set the TCP write-side high water mark. TCP will exert flow control back to the Stream head when more than this amount of data has been sent downstream to TCP but has not yet been transmitted.

Default: The tunable parameter *tcp_xmit_hiwater_de*f, whose default value is 32768.

`XTI_SNDLOWAT`	**level: `XTI_GENERIC`**
(Inherited)	**len: sizeof(long)**

`XTI_SNDLOWAT` is used to set the TCP write-side low water mark. Because of how TCP implements flow control, *q_lowat* itself is not set to this value. However, once TCP exerts flow control, it will remain in effect until the number of bytes not sent s less than the value set by this option.

Default: The tunable parameter *tcp_xmit_lowater_de*f, whose default value is 8192.

`XTI_SND_COPYAVOID`	**level: `XTI_GENERIC`**
(Inherited)	**len: sizeof(long)**

This option is simply passed through to IP. This option is simply recorded by IP; it has no effect.

Default: Off.

TPI Primitives for UDP

T_ADDR_REQ and T_ADDR_ACK

As indicated by the *XPG4_1* flag in the *PROVIDER_flag* field of the **`T_INFO_ACK`**, UDP supports the **`T_ADDR_REQ`** primitive. The local address field in the **`T_ADDR_ACK`** is non-zero only if the stream is bound. The remote address field is nonzero only if a **`T_CONN_REQ`** has been issued on the stream.

NOTE

If an endpoint has been bound to **`INADDR_ANY`**, the IP address portion of the returned local address will be zero; the length will be *sizeof(struct sockaddr)*, which is 16.

T_BIND_REQ and T_BIND_ACK

A **T_BIND_REQ** may be issued as the normal TPI M_PROTO message, or it may be specified as an **I_STR** ioctl with *ioc_cmd* field equal to **TI_BIND** (defined in **timod.h**), and the *ioc_dp* buffer containing the **T_BIND_REQ** request as it would appear in an M_PROTO message. The ioctl version allows UDP to check privilege for the request based on the current credentials of the stream, not those at the time the stream was opened. This may be relevant for applications which perform a "set uid" to bind to a privileged port (less than 1024) after the stream was opened.

If the *ADDR_length* field in the **T_BIND_REQ** is 0, the request is interpreted as a bind for **INADDR_ANY**.

If the *ADDR_length* field is nonzero, it must be 8 or 16, and the address specified at *ADDR_offset* must be a *sockaddr_in* structure as shown below. Even though "correct" applications will use an address length of 16, equal to *sizeof(struct sockaddr)*, a length of 8 is also supported for applications not including the zero pad in the length count. The *sin_port* and *sin_addr* fields must be specified in Network Order (Big Endian); the *sin_family* field must be **AF_INET** or zero.

```
struct sockaddr_in {
    short           sin_family;
    ushort          sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

If *sin_addr* is a Class D multicast address, then only multicast frames for that address will be delivered. However, the **T_BIND_REQ** itself does not enable any multicast address. Multicasts are enabled by UDP options; see *"UDP Options for IP Multicast" on page 2-48* for details.

Binding to Addresses and Ports Already in Use

Table 2-3 summarizes the conditions under which a **T_BIND_REQ** sent to UDP will succeed or fail, depending upon the setting of **IP_REUSEADDR** and **IP_REUSEPORT** and the specified IP address. In all cases, the **T_BIND_REQ** is specifying the same UDP port number as a port already in use by another UDP instance. *NewUDP* is the stream on which the current **T_BIND_REQ** is being sent; *OldUDP* is some stream previously bound to the same UDP port. *IP1* and *IP2* are two specific, distinct, non-multicast IP addresses; ClassD is a multicast address; ANY is **INADDR_ANY (0.0.0.0)**.The rows across the top show eight different combinations of setting these options. *RA* stands for **IP_REUSEADDR**; *RP* stands for **IP_REUSEPORT**. An *X* in a cell means the **T_BIND_REQ** will succeed; an empty cell means the bind will fail.

Table 2-3. Binding to Addresses and Ports Already in Use

RA for NewUDP RP for NewUDP RP for OldUDP		off off off	off off on	off on off	off on on	on off off	on off on	on on off	on on on
OldUDP	NewUDP								
IP1	IP1				X				X
IP1	IP2	X	X	X	X	X	X	X	X
IP1	ANY				X	X	X	X	X
IP1	ClassD	X	X	X	X	X	X	X	X
ANY	IP1				X	X	X	X	X
ANY	ANY				X				X
ANY	ClassD				X		X		X
ClassD	IP1	X	X	X	X	X	X	X	X
ClassD	ANY				X	X	X	X	X
ClassD	ClassD				X		X		X

In addition, note that:

- If the *NewUDP* is binding to a ClassD address and if **`IP_REUSEADDR`** is set for either stream, then **`IP_REUSEPORT`** is also considered set for that same stream. In this case, each stream bound to the same ClassD address and port will receive a copy of inbound packets addressed to that ClassD address and port.
- **`IP_REUSEPORT`** is "cooperative", that is, both streams must have it set for it to apply. When set, all binds succeed.

T_BIND_REQ "Address in use" Errors

Whenever the process of the previous paragraph results in an IP address/port number being unavailable, the TPI document dictates that the provider must return a `TADDRBUSY` error. Some early TLI libraries relied upon the provider returning a **`T_BIND_ACK`** with a different port than requested rather than a **`T_ERROR_ACK`**. In accordance with UNIX System V TLI convention. UDP will return a **`T_BIND_ACK`** with a different (unused) port than was specified in the original **`T_BIND_REQ`**.

Broadcast Binds

A UDP stream may bind to an "all 1's" broadcast or subnet broadcast address. This feature exists for specialized "read only" applications. However, if a datagram is sent on this stream, UDP will fill in a proper non-broadcast source address depending upon the interface used to send the datagram to the destination.

T_CONN_REQ and T_CONN_CON

In addition to the normal connection-less TPI primitives, UDP supports the concept of a "connected UDP" stream. This permits socket-style **`read`** and **`write`** system calls to be used with UDP. Once connected, all subsequent **`write`** calls will send data to the specified destination address. Furthermore, only packets from the remote (connected) address will be delivered to this stream. In this case, these packets will be sent upstream as `M_DATA` messages which can then be directly read using the **`read`** system call.

Sending a **`T_UNITDATA_REQ`** using **`putmsg`** is the only mechanism that is guaranteed to preserve message boundaries.

If the **`T_CONN_REQ`** is properly formatted, UDP will return both a `T_OK_ACK` and a `T_CONN_CON` message to satisfy TPI semantics. Note, however, that no verification of IP destination address has been made, so that if the address is in error, subsequent **`write`** operations will appear to succeed. Only if the error causes an ICMP error to be generated, and the UDP tunable parameter *udp_pass_up_icmp* or UDP option **`UDP_RX_ICMP T_DISCON_REQ`** has been set, will a `T_UDERROR_IND` message be delivered. Furthermore, since the remote address is incorrect, no inbound datagrams will be delivered by IP to this UDP stream.

If a **`T_UNITDATA_REQ`** is sent downstream, UDP will process this normally, ignoring the destination from the previous **`T_CONN_REQ`**.

If the destination address in the **`T_CONN_REQ`** is **`INADDR_ANY`**, datagrams will be sent to the local loopback address, 127.0.0.1.

Any options specified in the **`T_CONN_REQ`** are quietly ignored.

T_DISCON_REQ

A **`T_DISCON_REQ`** may be issued on a connected UDP stream to remove the association established by the **`T_CONN_REQ`**. Subsequently, the stream remains open and will receive packets from any source directed to the bound UDP port. In addition, if the **`UDP_RX_ICMP`** option had been set to 2 (BSD semantics for passing ICMP messages upstream), the option is disabled an no ICMP are sent. Note that the option is reset only if it had been explicitly set to 2.

T_INFO_REQ and T_INFO_ACK

Version 1.5 of the TPI document introduced a new *XPG4_1 flag* in the *PROVIDER_flags* field of the **`T_INFO_ACK`** structure. This flag is set if it is defined in the current implementation. Table 2-4 shows the values returned by UDP.

Table 2-4. UDP Returned Values

Field	Return Value
TSDU_size	`65508`
ETSDU_size	`-2`
CDATA_size	`-2`
DDATA_size	`-2`
ADDR_size	`16`
OPT_size	`256`
TIDU_size	`65508`
SERV_type	`T_CLTS`
CURRENT_state	`TPI state of UDP`
PROVIDER_flag	`SENDZERO | XPG4_1`

T_UNBIND_REQ

After a successful **`T_UNBIND_REQ`**, datagrams will not be delivered upstream. In addition, the effect of any **`T_CONN_REQ`** will be cancelled.

T_UNITDATA_REQ and T_UNITDATA_IND

Options specified on a **`T_UNITDATA_REQ`** override option settings for the single datagram. The following XTI options are supported on a **`T_UNITDATA_REQ`**:

```
IP_DONTROUTE          IP_OPTIONS           IP_TTL
  IP_TOS             UDP_CHECKSUM
```

The format for specifying options is the same as used with a **`T_OPTMGMT_REQ`** with **`MGMT_flags`** equal to **`T_NEGOTIATE`**. See *"UDP Options" on page 2-44* for an explanation of these options.

Options are inserted into `T_UNITDATA_IND` messages only if they have been requested by previously setting the corresponding option as shown in Table 2-5.

Note that use of **`T_CONN_REQ`** has no impact on the behavior of subsequent **`T_UNITDATA_REQ`** primitives, i.e., the address and options on the **`T_UNITDATA_REQ`** take precedence. On the other hand, if a UDP stream is connected, **`T_UNITDATA_IND`** primitives will never be sent upstream; rather only `M_DATA` messages (from the connected address) will be sent upstream.

If the destination address is **`INADDR_ANY`**, either in the **`T_UNITDATA_REQ`** or as set with a **`T_CONN_REQ`**, the datagram will be sent to the local loopback address, 127.0.0.1.

Table 2-5. Options Inserted Into T_UNITDATA_IND Messages

Send a **T_OPTMGMT_REQ** for this Option	To receive T_UNITDATA_IND messages containing this Option
IP_RECVDSTADDR	**IP_RECVDSTADDR** with the IP destination address.
IP_RECVIFADDR	**IP_RECVIFADDR** with the IP address of the interface on which the datagram was received.
IP_RECVOPTS	**IP_OPTIONS** with IP header.

UDP Options

UDP supports the following options, listed alphabetically by their XTI name; options for IP multicast are listed separately in *UDP Options for IP Multicast* and *UDP Options for IP Multicast* sections in this chapter. See *"Options Management" on page 2-57* for details on **T_OPTMGMT_REQ** formats; see *"TLI Option Names" on page 2-59* for a mapping of TLI (socket) option names to XTI name. Boolean values are true/on/yes (nonzero) and false/off/no (zero); by convention with sockets, a nonzero value of some boolean options is the value of the option name.

IP_BROADCAST — **level: INET_IP**
len: sizeof(long)

This option is simply recorded by UDP and IP. It has no effect.

Default: Off.

IP_BROADCAST_IF — **level: INET_IP**
IP_BROADCAST_IFNAME — **len: —**

These options are provided to support the UDP-based protocol DHCP (Dynamic Host Configuration Protocol). DHCP requires a client to engage in a UDP protocol with a source address of 0.0.0.0 until sufficient configuration information is exchanged. Normally, broadcasts are used over the interface until the interface is configured. To permit one interface to be DHCP-configured, while others are fully operational, DHCP client applications must be able to specify which interface to broadcast requests from. These options provide this capability. Note that:

1. Only a single 0.0.0.0 interface may exist at any time. Therefore, a DHCP client which wishes to bring up multiple interfaces will need to process them serially and use the **IP_BROADCAST_IF[NAME]** option to specify the current one. This serialization is only necessary before the interface address is established; multiple DHCP conversations may proceed in parallel once the IP address is set.
2. During DHCP initialization, IP will accept and send upstream non-broadcast packets for DHCP/BOOTP destination ports for an

unknown IP address (normally IP discards or forwards such packets). This permits IP to work with BOOTP/DHCP servers which don't use broadcasts.

3. A DHCP client is not supplied with PowerMAX OS.

For convenience there are two options with which an application may specify a particular interface to use for a broadcast. **IP_BROADCAST_IFNAME** takes a string containing an interface name such as "le0" (as returned by **SIOCGIFCONF** ioctl, for example); **len** is the length of the NULL-terminated string, which must be **ð** **IFNAMSIZ**. **IP_BROADCAST_IF** is passed a four byte IP address in Network (Big Endian) byte order (as returned by an **IP_RECVIFADDR** option, for example); **len** equals 4.

Default: Off. Broadcasts are sent out all interfaces which match the broadcast address; e.g., all 1's will go out all interfaces; a subnet broadcast will go out all interfaces with matching subnet address.

IP_DONTROUTE **level: INET_IP**
len: sizeof(long)

The value of this option is used by UDP to set the TTL in the IP header of outbound datagrams to 1, causing packets not to be forwarded through a router.

Default: Off.

IP_OPTIONS **level: INET_IP**
len: —

For **T_NEGOTIATE** the argument contains an IP option string exactly as it would appear in an IP packet, e.g., data in network (Big Endian) byte order. **len** is rounded up to a multiple of 4 and the argument is padded with zeros, if necessary. If source routing is present, the first hop is inserted into the IP header destination field and the pointer moved down one. The option is copied into the IP packet immediately after the IP simple header, i.e., starting at byte offset 20.

Default: No options.

IP_RECVDSTADDR **level: INET_IP**
len: sizeof(long)

When set, T_UNITDATA_IND messages will contain an **IP_RECVDSTADDR** option with value equal to the destination address from the IP header for this datagram, in Network (Big Endian) order.

Default: Off.

IP_RECVIFADDR **level: INET_IP**
len: sizeof(long)

When set, T_UNITDATA_IND messages will contain an **IP_RECVIFADDR** option with value equal to the IP address of the interface on which this datagram was received, in Network (Big Endian) order.

Default: Off.

IP_RECVOPTS	**level: INET_IP** **len: sizeof(long)**

When set, T_UNITDATA_IND messages will contain an **IP_OPTIONS** option with value equal to the IP header options (starting at offset 20 of the IP datagram) of the packet containing this datagram.

Default: Off.

IP_REUSEADDR **IP_REUSEPORT**	**level: INET_IP** **len: sizeof(long)**

See *“T_BIND_REQ and T_BIND_ACK” on page 2-29* for a discussion of these options.

Default: Off.

IP_TOS	**level: INET_IP** **len: sizeof(char)**

The option value is inserted, without verification, into the *TOS* field of all subsequent IP packets.

Default: Zero.

IP_TTL	**level: INET_IP** **len: sizeof(char)**

The option value is inserted, without verification, into the *TTL* field of all subsequent IP packets.

Default: Tunable parameter *udp_def_ttl* whose default value is 255.

UDP_CHECKSUM	**level: INET_UDP** **len: sizeof(long)**

Set the option value to zero to disable UDP data checksums; set to nonzero to include UDP data checksums.

Default: On. Data checksums are computed.

UDP_RX_ICMP	**level: INET_UDP** **len: sizeof(char)**

RFC 1122 says that UDP MUST pass all ICMP errors up to the application. We implement this by sending the error upstream in a **T_UDERROR_IND**. Of course this breaks many applications which are not expecting to see these messages. Therefore, Mentat UDP added a tunable parameter *udp_pass_up_icmp* which controls whether or not ICMP errors are delivered upstream to the application; 0 for no, or 1 for yes. The setting of the tunable parameter may be overridden on a per-stream basis with the **UDP_RX_ICMP** option.

Note that the default behavior for BSD implementations is always to send errors up on connected UDP sockets (streams) and never on unconnected UDP sockets. This is accommodated by setting *udp_pass_up_icmp* to 2.

Default: Depends on the setting of *udp_pass_up_icmp*, which by default is 1.

XTI_DEBUG — **level: XTI_GENERIC**
len: sizeof(long)

This option is simply recorded by UDP; it has no effect.

Default: Off.

XTI_DISTRIBUTE — **level: XTI_GENERIC**
len: sizeof(long)

The **XTI_DISTRIBUTE** option permits multiple UDP streams bound to the same local address and port to receive UDP packets in a round-robin fashion. Some UDP-based applications on multiprocessor systems may attain higher performance if they are able to distribute read and write operations across multiple UDP streams.

This option must be issued before the stream is bound. Note that **IP_REUSEADDR** and **IP_REUSEPORT** must also be specified (before the bind request) to enable this feature.

Default: Off.

XTI_PROTOTYPE — **level: XTI_GENERIC**
len: sizeof(long)

This option is provided for compatibility with other modules, notably RAWIP, that support multiple protocols; it has no effect in UDP. Only **IPPROTO_UDP** may be set in a **T_NEGOTIATE** request.

Default: **IPPROTO_UDP**.

XTI_RCVBUF — **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the read-side *queue q_hiwat*. Since UDP has no read-side service routine, this option is effectively a no-op.

Default: 512.

XTI_RCVLOWAT — **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the read-side *queue q_lowat*. Since UDP has no read-side service routine, this option is effectively a no-op.

Default: 128.

XTI_SNDBUF — **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the write-side *queue q_hiwat.* Since UDP has no write-side service routine, this option is effectively a no-op.

Default: 512.

`XTI_SNDLOWAT` **level: `XTI_GENERIC`**
len: sizeof(long)

The value in the option sets the write-side *queue q_lowat*. Since UDP has no write-side service routine, this option is effectively a no-op.

Default: 128.

UDP Options for IP Multicast

This section describes UDP option which an application may use to join a multicast group. These same options are also supported by RAWIP (refer to *TPI Primitives for RAWIP* section in this chapter). Options used by `mrouted` for multicast routing are also RAWIP based. (Refer to *"RAWIP Options for Multicast Routing" on page 2-55* for more information.)

With UDP it is possible for multiple applications to receive copies of the same multicast datagrams. See *"T_BIND_REQ and T_BIND_ACK" on page 2-29* for a discussion of **`IP_REUSEADDR`** and **`IP_RESUEPORT`** options and **`T_BIND_REQ`** for details.

`IP_ADD_MEMBERSHIP` **level: `INET_IP`**
len: sizeof(struct ip_mreq)

This option is used by applications to join a host group. The option value is an *ip_mreq* structure, defined in **`in.h`** and shown below:

```
struct ip_mreq {
    struct in_addr      imr_multiaddr;
    struct in_addr      imr_interface;
}
```

imr_multiaddr is the Class D address of the group to join, and *imr_interface* the IP address of the interface on which to join.

Since host group membership is on a per interface basis, the application can specify on which interface it wants to join. An application can join the same multicast group on multiple interfaces. To use the default multicast interface, set *imr_interface == INADDR_ANY.*

The group address is used to determine which destination multicasts are to be received by the application. The "add membership" information is used by UDP to determine whether a given UDP stream wants a packet. This check only looks for multicast group membership. If multiple applications are sharing a multicast group address, they must specify the same interface; otherwise an application may receive packets from interfaces other than the one it specified.

`IP_DROP_MEMBERSHIP` **level: `INET_IP`**
len: sizeof(struct ip_mreq)

This option is used to leave a group. `imr_multiaddr` is the name (Class D address) of the group to leave. *imr_interface* the IP address of the interface which must match an interface on which the group was joined, or **`INADDR_ANY`** to leave the group on the default interface.

`IP_MULTICAST_IF` **level: `INET_IP`**
len: sizeof(long)

A sender can use this option to select the interface to use for outgoing multicast datagrams. Normally this is not necessary since hosts will have a route to the default interface for 224.0.0.0. However, some applications may wish to multicast on all attached subnetworks. Such applications must loop through all multicast capable interfaces and use this option to select each interface in turn before sending the multicasts. The argument is the IP address of the interface in Network (Big Endian) byte order.

`IP_MULTICAST_LOOP` **level: `INET_IP`**
len: sizeof(char)

By default IP will loop back multicast datagrams to any member on the sending machine. To disable this behavior, pass a value of zero for this option.

Default: On.

`IP_MULTICAST_TTL` **level: `INET_IP`**
len: sizeof(char)

If no other control is performed, a multicast packet will be sent with an IP time-to-live (TTL) of 1, i.e., it will not be forwarded by any multicast routers. The value of this option is used in the TTL field of the IP header in all subsequent multicast packets.

Default: 1.

TPI Primitives for RAWIP

The RAWIP module provides a TPI interface directly to IP. There are two basic modes of operation:

- Data passed to the RAWIP module contains an IP header which is passed directly to IP.
- Data passed to the RAWIP module has no IP header and RAWIP constructs a header before passing the data to IP.

The mode is determined from the protocol family which is specified when the stream is bound. For **`IPPROTO_RAWIP`** and **`IPPROTO_IGMP`**, RAWIP assumes that IP headers are in the data; for other protocols, including **`IPPROTO_ICMP`**, RAWIP creates the IP header. This behavior can be overridden using the **`IP_HDRINCL`** option.

When RAWIP is opened, the default protocol is **`IPPROTO_ICMP`**. This may be changed with the **`XTI_PROTOTYPE`** option. The protocol may also be specified when a stream is bound.

T_BIND_REQ and T_BIND_ACK

A **`T_BIND_REQ`** may be issued as the normal TPI `M_PROTO` message, or it may be specified as an **`I_STR`** ioctl with *ioc_cmd* field equal to **`TI_BIND`** (defined in *timod.h*), and the *ioc_dp* buffer containing the **`T_BIND_REQ`** request as it would appear in an `M_PROTO` message. The ioctl version allows RAWIP to check privilege for the request based on the current credentials of the stream, not those at the time the stream was opened.

For RAWIP, a **`T_BIND_REQ`** is used to bind to a protocol, an IP address, or both. In all three cases, a **`T_BIND_ACK`** is returned on success; its fields are unchanged from the **`T_BIND_REQ`**.

Binding to a Protocol

To specify a protocol, set the *ADDR_length* field to 1 and place the protocol number in the byte at offset *ADDR_offset*. Specifying **`IPPROTO_RAW`** or **`IPPROTO_IGMP`** implicitly sets the **`IP_HDRINCL`** option, so RAWIP will assume all data messages begin with a valid IP header.

Only privileged processes may bind to a protocol other than **`IPPROTO_ICMP`**. Only one stream may bind to **`IPPROTO_TCP.`** Thus if TCP is configured, an attempt to bind to **`IPPROTO_TCP`** will fail. For all other protocols, each stream bound to a specific protocol will receive copies of IP datagrams for that protocol.

RAWIP permits a protocol bind in any state. If the stream is already bound to a protocol, the new **`T_BIND_REQ`** will simply be a “re-bind” to the new protocol.

Binding to an IP Address

A RAWIP stream may bind to an IP address. However, such a bind serves only to verify that the specified address is a valid local address for this host; otherwise, RAWIP ignores the IP address.

If a **`T_BIND_REQ`** specifies an IP address and *ADDR_length* of 16 *(== sizeof(struct sockadr_in))*, RAWIP will assume this is a “re-bind” to the current protocol. The address is given in a *sockaddr_in* structurestruct *sockaddr_in* as shown below:

```
{
short           sin_family;
ushort          sin_port;
struct in_addr  sin_addr;
char            sin_zero[8];
};
```

A single **`T_BIND_REQ`** may be used to specify both an IP address and a protocol. Specify *ADDR_length* equal to 17 *(== sizeof(struct sockaddr_in) + 1)*. Place the IP address in a *sockaddr_in* structure at *ADDR_offset*; in the next byte after the *sockaddr_in structure* place the protocol byte as shown in Figure 2-2.

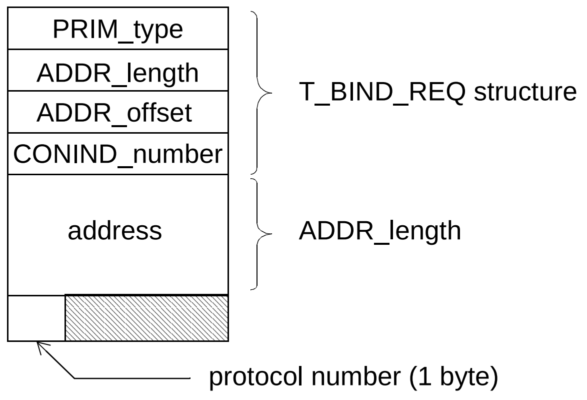

Figure 2-2. Message Block Containing T_BIND_REQ Message Sent to IP

If the *ADDR_length* field in the **T_BIND_REQ** is 0, the request is interpreted as a bind for **INADDR_ANY**. In fact, a protocol bind is equivalent to an *ADDR_length == 17 bind*, with IP address set to **INADDR_ANY**.

The *sin_port* and *sin_addr* fields must be specified in Network Order (Big Endian); the *sin_family* field must be *AF_INET* or zero.

T_CONN_REQ and T_CONN_CON

A **T_CONN_REQ** may be used to specify a destination address to insert in front of data sent down stream with a *write* system call, i.e., as an M_DATA message. RAWIP will create a simple IP header (no options) from the destination address in the **T_CONN_REQ** and the bound address of the stream. TTL is set to the system default, the tunable parameter *rawip_ttl_default*, or as set by the **IP_TTL** option.

Any options supplied on the **T_CONN_REQ** are quietly ignored. To specify options, use the **IP_OPTIONS** option (refer to *"RAWIP Options" on page 2-53*).

A successful "connect" results in a **T_OK_ACK** for the **T_CONN_REQ**, followed by a **T_CONN_CON** containing the specified destination address and no options.

Unlike "connected UDP" (refer to the *"T_CONN_REQ and T_CONN_CON" on page 2-42*), RAWIP does no filtering of inbound packets based on the destination of the **T_CONN_REQ**.

T_DISCON_REQ

A **T_DISCON_REQ** may be issued on a connected RAWIP stream to remove the association established by the **T_CONN_REQ**. Subsequently, the stream remains open, but M_DATA messages sent downstream will be silently discarded.

T_INFO_REQ and T_INFO_ACK

Version 1.5 of TPI document introduced a new *XPG4_1* flag in the *PROVIDER_flags* field of the **`T_INFO_ACK`** structure. This flag is set if it is defined in the current implementation. Table 2-6 shows the values returned by RAWIP:

Table 2-6. RAWIP Values

Field	Return Value
TSDU_size	`65536`
ETSDU_size	`-2`
CDATA_size	`-2`
DDATA_size	`-2`
ADDR_size	`16`
OPT_size	`64`
TIDU_size	`65536`
SERV_type	`T_CLTS`
CURRENT_state	`TPI state of RAWIP`
PROVIDER_flag	`0`

T_UNBIND_REQ

After a **`T_UNBIND_ACK`**, no messages will be sent upstream. However, the protocol from the last bind operation is remembered, and a subsequent bind which doesn't explicitly state a protocol will use the same protocol.

T_UNITDATA_REQ and M_DATA

For sending data, the RAWIP module distinguishes between two cases:

1. The data buffer includes the full IP header for the datagram.
2. The data buffer contains only data and RAWIP must insert an IP header in front of it.

If the stream is bound to **`IPPROTO_RAW`** or **`IPPROTO_IGMP`**, or the **`IP_HDRINCL`** option has been set, the data is assumed to contain a full IP header in the initial bytes; RAWIP does not add its own header. In all other cases, RAWIP inserts an IP header in front of the data.

If the data is assumed to contain IP headers, `M_DATA` messages are passed to IP without change, except:

- The header is initialized for IP checksum calculation.
- Source route options are processed to set the destination address to the first hop in the source route option.

If headers are not assumed to be in the data, `M_DATA` messages are valid on connected RAWIP streams. In this case the destination address and any IP options from the **`T_CONN_REQ`** will be inserted in front of the data.

Options specified on a **`T_UNITDATA_REQ`** are ignored. If RAWIP must add a header that includes options, these options must be specified separately in advance by sending a **`T_OPTMGMT_REQ`** with **`IP_OPTIONS`** which contains the full IP header to use.

T_UNITDATA_IND

`T_UNITDATA_IND` messages are created for all inbound data, even for a "connected **`RAWIP`**" stream. The source address in the **`T_UNITDATA_IND`** is the IP address of the source (no port). No options are returned; instead, the entire IP header is returned in the first `M_DATA` message block.

The value returned in the length field of the IP header may or may not include the length of the IP header itself, depending upon the setting of the tunable parameter *rawip_bsd_compat*. If *rawip_bsd_compat* is nonzero (the default), RAWIP excludes the length of the IP header itself; if *rawip_bsd_compat* is zero, the IP header length is included.

RAWIP Options

RAWIP supports the following options, listed alphabetically by their XTI name. See *"Options Management" on page 2-57* for details on **`T_OPTMGMT_REQ`** formats; see *"TLI Option Names" on page 2-59* for a mapping of TLI (socket) option names to XTI name. Boolean values are true/on/yes (nonzero) and false/off/no (zero); by convention with sockets, nonzero values of some boolean values is the value of the option name.

`IP_BROADCAST` **level: `INET_IP`**
len: sizeof(long)

This option is simply recorded by RAWIP and IP. It has no effect.

Default: Off.

`IP_DONTROUTE` **level: `INET_IP`**
len: sizeof(long)

The value of this option is used by RAWIP to set the TTL in the IP header of outbound datagrams to 1, causing packets not to be forwarded through a router.

Default: Off.

IP_HDRINCL **level: INET_IP**
len: sizeof(long)

When set, RAWIP assumes M_DATA and T_UNITDATA_REQ messages start with the IP header to use for sending the datagram. Otherwise RAWIP will insert am IP header in front of the data.

Default: For **IPPROTO_RAW** and **IPPROTO_IGMP**, headers are assumed to be included; otherwise not.

IP_OPTIONS **level: INET_IP**
len: —

This option is only meaningful if **IP_HDRINCL** is off, i.e., RAWIP creates the IP header. When RAWIP is creating the IP header, this option is the only mechanism for inserting IP options into the IP header.

For **T_NEGOTIATE**, the argument contains an IP options specification exactly as it would appear in an IP packet, e.g., data in Network (Big Endian) byte order. **len** is rounded up to a multiple of 4 and the argument is padded with zeros, if necessary. If the source routing is present, the first hop is inserted into IP header destination field and the hop pointer is advanced. The option is copied into the IP packet immediately after the IP simple header, i.e., starting at byte offset 20.

IP_OPTIONS cannot be read with a **T_OPTMGMT_REQ**; options are returned with each **T_UNITDATA_IND**.

Default: No options.

IP_TOS **level: INET_IP**
len: sizeof(char)

This option is passed through to IP. The value is inserted, without verification, into the TOS field of all subsequent IP packets.

Default: Zero.

IP_TTL **level: INET_IP**
len: sizeof(char)

This option is passed through to IP. The value is inserted, without verification, into the TTL field of all subsequent IP packets.

Default: Tunable parameter *rawip_def_ttl* whose default value is 255.

XTI_DEBUG **level: XTI_GENERIC**
len: sizeof(long)

This option is simply recorded by RAWIP; it has no effect.

Default: Off.

XTI_PROTOTYPE **level: XTI_GENERIC**
len: sizeof(long)

This option may be used to set the protocol for the RAWIP stream. The option value is the protocol number. When RAWIP is opened, the protocol is set to **IPPROTO_ICMP**. To set other protocols, the stream must be privileged.

See **T_BIND_REQ** (*T_BIND_REQ and T_BIND_ACK* section in this chapter) for other information about protocols with RAWIP.

XTI_RCVBUF **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the read-side queue *q_hiwat*. Since RAWIP has no read-side service routine, this option is effectively a no-op.

Default: 512.

XTI_RCVLOWAT **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the read-side *<token>queue q_lowat*. Since RAWIP has no read-side service routine, this option is effectively a no-op.

Default: 128.

XTI_SNDBUF **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the write-side queue *q_hiwat*. Since RAWIP has no write-side service routine, this option is effectively a no-op.

Default: 512.

XTI_SNDLOWAT **level: XTI_GENERIC**
len: sizeof(long)

The value in the option sets the write-side *queue q_lowat*. Since RAWIP has no write-side service routine, this option is effectively a no-op.

Default: 128.

RAWIP Options for Multicast Routing

The following options are generally used only by multicast routers, such as UNIX’s `mrouted` multicast routing daemon program. Note: `mrouted` is not supplied in Power-MAX OS.

MRT_ASSERT **level: INET_IP**
len: sizeof(long)

Controls PIM Assert processing. Option value is 1 to enable, 0 to disable.

`MRT_ADD_MFC` **level: `INET_IP`**
len: sizeof(struct mfcctl)

The “value” of the option is a *mfcctl* structure as shown below:

```
struct mfcctl {
    struct in_addr  mfcc_origin;
    struct in_addr  mfcc_mcastgrp;
    vifi_t          mfcc_parent;
    uchar           mfcc_ttls[MAXVIFS];
};
```

`MRT_ADD_VIF` **level: `INET_IP`**
len: sizeof(struct vifctl)

The “value” of the option is a *vifctl* structure as shown below:

```
struct vifctl {
    vifi_t          vifc_vifi;
    uchar           vifc_flags;
    uchar           vifc_threshold;
    uint            vifc_rate_limit;
    struct in_addr  vifc_lcl_addr;
    struct in_addr  vifc_rmt_addr;
};
```

`MRT_DEL_MFC` **level: `INET_IP`**
len: sizeof(struct mfcctl)

The “value” of the option is a *mfcctl* structure as shown below:

```
struct mfcctl {
    struct in_addr  mfcc_origin;
    struct in_addr  mfcc_mcastgrp;
    vifi_t          mfcc_parent;
    uchar           mfcc_ttls[MAXVIFS];
};
```

`MRT_DEL_VIF` **level: `INET_IP`**
len: sizeof(vifi_t)

The argument is the index of the *vif* to delete; same as *vifc_vifi* argument of **`MRT_ADD_VIF`**. The *vifi_t type* is defined as unsigned short.

`MRT_DONE` **level: `INET_IP`**
len: 0

Terminate multicast router service. There is no value passed with this option.

`MRT_INIT` **level: `INET_IP`**
len: sizeof(int)

Initialize multicast router service. The value must be the integer 1.

`MRT_VERSION`	**level: `INET_IP`** **len: sizeof(long)**

This is a read-only option. The value returned is the DVMRP version number. The correct version, 3.5, is returned as the hexadecimal value 0x0305.

RAWIP Options for IP Multicast

The multicast options described in the UDP Options for IP Multicast section of this chapter for UDP may also be used with RAWIP. These options are simply passed through to IP; RAWIP takes no action on any of them.

Options Management

Previous sections have listed the options supported by each of TCP, UDP, and RAWIP. This section describes how to specify an option in a **`T_OPTMGMT_REQ`**. Since PowerMAX OS TCP/IP modules use common code for options management, the discussion which follows applies equally to TCP, UDP, and RAWIP.

Semantics of T_OPTMGMT_REQ Message Formats

Although the TPI document is precise about the rules for **`T_OPTMGMT_REQ`** message formats, it says nothing about the internal formats of options themselves. PowerMAX OS TCP/IP uses standard UNIX System V semantics for **`T_OPTMGMT_REQ`** message formats.

MGMT_flags Field in T_OPTMGMT_REQ

The *MGMT_flags* field of the **T_OPTMGMT_REQ** structure defines the request. The values shown in Table 2-7 are recognized.

Table 2-7. MGMT_flags Field

MGMT_flags	Description
T_CHECK	For each option level, name, and value, confirm that the provider would successfully allow this option to be set via a **T__NEGOTIATE**.
T_CURRENT	For each option level and name, return the current value.
T_DEFAULT	For each option level and name, return the provider's default value.
T_NEGOTIATE	For each option level, name, and value, set the specified option to the supplied value.

TLI Semantics for Options Processing

TLI options are specified in an `opthdr` structure, defined in **sys/socket.h** and shown below:

```
struct opthdr {
    long        level;
    long        name;
    long        len;
};
```

A **T_OPTMGMT_REQ** can contain zero or more opthdr structures and associated values, as shown in Figure 2-3. The `opthdr` structures are long word aligned by padding (if necessary) after the option value. *OPT_offset* is the offset from the start of the message block containing the **T_OPTMGMT_REQ** structure to the start of the array of `opthdr` structures; *OPT_length* is the total length of all `opthdr` structures and values and padding. The *len* field in each `opthdr` structure is the length of the option value (excluding any padding) in bytes.

If a request contains several `opthdr` structures, and one or more fail, then the entire **T_OPTMGMT_REQ** will fail. This results from the fact the `opthdr` structure lacks a status field, so no mechanism exists to return different results for different options in the same request.

PowerMAX OS TCP/IP options processing is subject to the following restrictions.

- **T_ALLOPT** is not supported.
- Consistent with existing SVR4 implementations, **T_CHECK** is equivalent to **T_CURRENT**, except that if an unrecognized option is given, **T_CHECK**

will return a **T_OPTMGMT_ACK** with **T_FAILURE**, while **T_CURRENT** will return an **T_ERROR_ACK** with **T_BADOPT**.

- For *MGMT_flags == T_NEGOTIATE, OPT_length* must be zero.

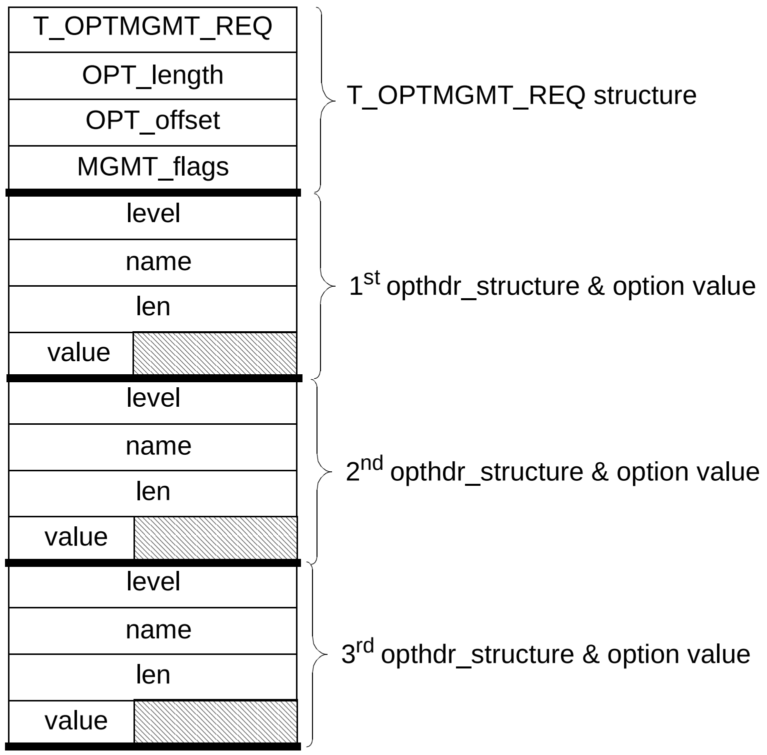
T_OPTMGMT_REQ
OPT_length
OPT_offset
MGMT_flags
level
name
len
value
T_OPTMGMT_REQ structure
1st opthdr_structure & option value
2nd opthdr_structure & option value
3rd opthdr_structure & option value

Figure 2-3. T_OPTMGMT_REQ with Three Options in TLI Format.

TLI Option Names

Table 2-8 list the TLI option names and levels, and the corresponding XTI names and levels. Refer to the description of the XTI option name in *X/Open Transport Interface (XTI)*[4] manual for a description of the option.

Table 2-8. TLI Option Names

TLI Option		XTI Option	
Name	Level	Name	Level
IP_ADD_MEMBERSHIP	**IPPROTO_IP**	**IP_ADD_MEMBERSHIP**	**INET_IP**
IP_BROADCAST_IF	**IPPROTO_IP**	**IP_BROADCAST_IF**	**INET_IP**
IP_BROADCAST_IFNAME	**IPPROTO_IP**	**IP_BROADCAST_IFNAME**	**INET_IP**

Table 2-8. TLI Option Names (Cont.)

TLI Option		XTI Option	
Name	Level	Name	Level
IP_DROP_MEMBERSHIP	**IPPROTO_IP**	**IP_DROP_MEMBERSHIP**	**INET_IP**
IP_HDRINCL	**IPPROTO_IP**	**IP_HDRINCL**	**INET_IP**
IP_LINK_STATUS	**IPPROTO_IP**	**IP_LINK_STATUS**	**INET_IP**
IP_MULTICAST_IF	**IPPROTO_IP**	**IP_MULTICAST_IF**	**INET_IP**
IP_MULTICAST_LOOP	**IPPROTO_IP**	**IP_MULTICAST_LOOP**	**INET_IP**
IP_MULTICAST_TTL	**IPPROTO_IP**	**IP_MULTICAST_TTL**	**INET_IP**
IP_OPTIONS	**IPPROTO_IP**	**IP_OPTIONS**	**INET_IP**
IP_RECVDSTADDR	**IPPROTO_IP**	**IP_RECVDSTADDR**	**INET_IP**
IP_RECVIFADDR	**IPPROTO_IP**	**IP_RECVIFADDR**	**INET_IP**
IP_RECVOPTS	**IPPROTO_IP**	**IP_RECVOPTS**	**INET_IP**
IP_TOS	**IPPROTO_IP**	**IP_TOS**	**INET_IP**
IP_TTL	**IPPROTO_IP**	**IP_TTL**	**INET_IP**
MRT_ADD_MFC	**IPPROTO_IP**	**MRT_ADD_MFC**	**INET_IP**
MRT_ADD_VIF	**IPPROTO_IP**	**MRT_ADD_VIF**	**INET_IP**
MRT_DEL_MFC	**IPPROTO_IP**	**MRT_DEL_MFC**	**INET_IP**
MRT_DEL_VIF	**IPPROTO_IP**	**MRT_DEL_VIF**	**INET_IP**
MRT_DONE	**IPPROTO_IP**	**MRT_DONE**	**INET_IP**
MRT_INIT	**IPPROTO_IP**	**MRT_INIT**	**INET_IP**
TCP_ABORT_THRESHOLD	**IPPROTO_TCP**	**TCP_ABORT_THRESHOLD**	**INET_TCP**
TCP_CONN_ABORT_THRESHOLD	**IPPROTO_TCP**	**TCP_CONN_ABORT_THRESHOLD**	**INET_TCP**
TCP_CONN_NOTIFY_THRESHOLD	**IPPROTO_TCP**	**TCP_CONN_NOTIFY_THRESHOLD**	**INET_TCP**
TCP_MAXSEG	**IPPROTO_TCP**	**TCP_MAXSEG**	**INET_TCP**
TCP_NODELAY	**IPPROTO_TCP**	**TCP_NODELAY**	**INET_TCP**
TCP_NOTIFY_THRESHOLD	**IPPROTO_TCP**	**TCP_NOTIFY_THRESHOLD**	**INET_TCP**
TCP_URGENT_PTR_TYPE	**IPPROTO_TCP**	**TCP_URGENT_PTR_TYPE**	**INET_TCP**
UDP_CHECKSUM	**IPPROTO_UDP**	**UDP_CHECKSUM**	**INET_UDP**
UDP_RX_ICMP	**IPPROTO_UDP**	**UDP_RX_ICMP**	**INET_UDP**
SO_BROADCAST	**SOL_SOCKET**	**IP_BROADCAST**	**INET_IP**
SO_DEBUG	**SOL_SOCKET**	**XTI_DEBUG**	**XTI_GENERIC**
SO_DISTRIBUTE	**SOL_SOCKET**	**XTI_DISTRIBUTE**	**XTI_GENERIC**

4. *X/Open Transport Interface* Document Number XO/CAE/91/600, X/Open Company Limited, January 1992.

Table 2-8. TLI Option Names (Cont.)

TLI Option		XTI Option	
Name	Level	Name	Level
SO_DONTROUTE	**SOL_SOCKET**	**IP_DONTROUTE**	**INET_IP**
SO_KEEPALIVE	**SOL_SOCKET**	**—**	**—**
SO_LINGER	**SOL_SOCKET**	**XTI_LINGER**	**XTI_GENERIC**
SO_OOBINLINE	**SOL_SOCKET**	**TCP_OOBINLINE**	**INET_TCP**
SO_PROTOTYPE	**SOL_SOCKET**	**XTI_PROTOTYPE**	**XTI_GENERIC**
SO_RCVBUF	**SOL_SOCKET**	**XTI_RCVBUF**	**XTI_GENERIC**
SO_RCVLOWAT	**SOL_SOCKET**	**XTI_RCVLOWAT**	**XTI_GENERIC**
SO_REUSEADDR	**SOL_SOCKET**	**IP_REUSEADDR**	**INET_IP**
SO_REUSEPORT	**SOL_SOCKET**	**IP_REUSEPORT**	**INET_IP**
SO_SNDBUF	**SOL_SOCKET**	**XTI_SNDBUF**	**XTI_GENERIC**
SO_SNDLOWAT	**SOL_SOCKET**	**XTI_SNDLOWAT**	**XTI_GENERIC**
SO_SND_COPYAVOID	**SOL_SOCKET**	**XTI_SND_COPYAVOID**	**XTI_GENERIC**
SO_TYPE	**SOL_SOCKET**	**—**	**—**
SO_USELOOPBACK	**SOL_SOCKET**	**—**	**—**

The following Socket options have no direct XTI equivalent.

SO_KEEPALIVE — **level: SOL_SOCKET**
len: sizeof(long)

This is the socket/TLI equivalent of the XTI **TCP_KEEPALIVE** option, but the argument and semantics differ. Even though it is **SOL_SOCKET** level, it is valid only for TCP.

If the value of the option is nonzero, keep-alive packets are sent; if zero, no keep-alive packets are sent. The actual value is ignored; the keep-alive interval is the tunable parameter *tcp_keepalive_interva*l, and the type is the tunable parameter *tcp_keepalives_kill*.

Default: Zero, no keep-alive packets are sent.

SO_TYPE — **level: SOL_SOCKET**
len: sizeof(long)

This is a “read only” option which returns the value **IPPROTO_TCP** for TCP, and **IPPROTO_UDP** for UDP. It is not supported for RAWIP.

SO_USELOOPBACK — **level: SOL_SOCKET**
len: sizeof(long)

This socket/TLI option is recognized by both TCP and UDP, However, this option is simply recorded; it has no effect.

Default:	Off.

3
IP Interfaces

Introduction 3-1
IP Module General Structure 3-1
    ILLs and Interfaces 3-3
    Loopback Device 3-4
Configuring an Interface 3-4
    IP Configuration ioctl’s 3-5
Defining IP Routes 3-14
    ARP ioctls. 3-18

3 IP Interfaces

Introduction

This chapter details the various interfaces to PowerMAX OS IP:

- ioctls for configuration,
- ioctls for managing routing tables

Two additional interfaces to IP are covered elsewhere: options passed through transport layer protocols to IP are described in Chapter 2 of this manual; IP's tunable parameters are described in Chapter 1.

IP Module General Structure

To understand better the various interfaces described in this chapter, we begin with a brief look at the relationship among IP, devices, and upper layer protocols.

IP is configured as both a module and a device. When an application opens a TCP, UDP, or RAWIP stream, in fact IP is opened as a device and the transport layer module is auto-pushed over IP as shown in Figure 3-1) We refer to this instance of IP as an "IP Client", or just IPC.

For each physical interface over which IP sends packets, one or two streams are created as part of IP configuration. If the device requires address resolution, e.g., an Ethernet card attached to a LAN, two streams will be created; one will have ARP above the device, the other will have ARP above IP above the device. This is shown in the left two streams of Figure 3-2.

For point-to-point interfaces, or other devices which require no address resolution, only a single stream is created with only the IP module pushed above the device. This is shown in the rightmost stream in Figure 3-2.

We refer to the instance of IP pushed over a device as an "IP Link Level" instance, or just ILL. Throughout the remainder of this chapter, we shall refer to IPC or ILL to identify the type of IP instance we are describing; when the distinction doesn't matter, we will simply refer to IP.

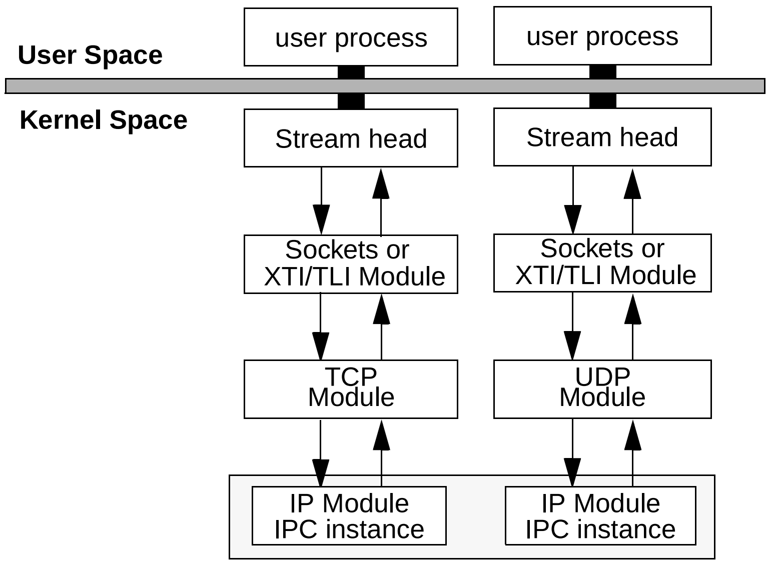

Figure 3-1. Configuration of IP as a Device Below TCP and UDP

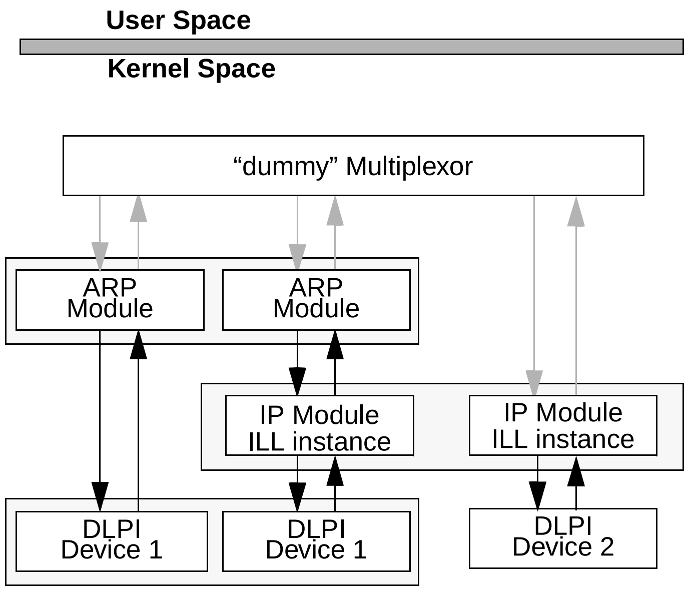

Figure 3-2. Configuration of STREAMS Devices, ARP, and ILL

Note that there is no direct relationship between any IPC instance and any ILL. Rather, inbound packets with a local destination are passed from the ILL to the module above the appropriate IPC instance, as determined by protocol bind information. Inbound packets with non-local IP addresses and outbound packets from an IPC are passed directly to the device below the appropriate ILL instance, as determined by IP's routing table.

ILLs and Interfaces

Each ILL is associated with a single stream to a device driver. Each driver stream is in turn associated with one or more IP addresses. Each IP address defines a logical IP interface, or IPIF. When we refer to "interface" in this chapter, we mean this logical interface or IPIF.

Each ILL instance includes a chain of IPIFs as shown in Figure 3-3. This figure shows two ILLs for devices "le0" and "le1". The "le0" device has a single IPIF for address 192.0.1.2; it would be referred to by name as "le0:0", or simply "le0". The "le1" device has three IPIFs corresponding to addresses 192.1.1.194, 192.1.1.193, and 192.1.1.129.

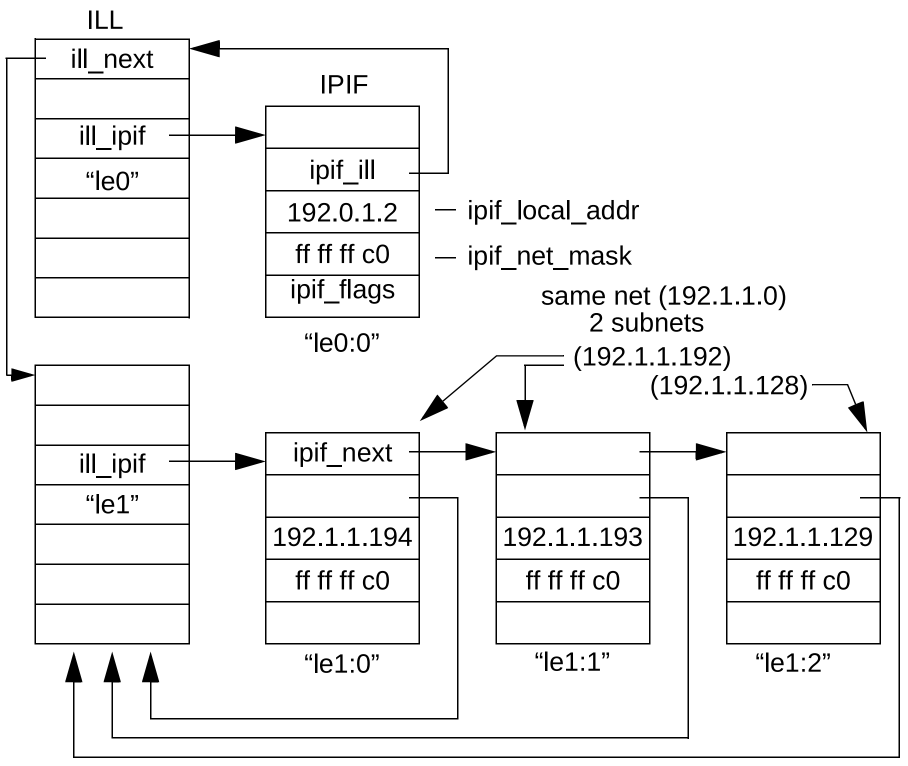

Figure 3-3. Relationship Between ILL's and IPIF's

IPIFs are created using configuration ioctls, which define attributes of the interface such as its local address, net mask, broadcast address, and status ("up" or "down"). See *"Configuring an Interface" on page 3-68* for details on configuring an interface.

To refer to the 192.1.1.193 interface, we use the name "`le1:1`". The "`le`" part of these names is taken from the name in the *qi_minfo->mi_idname* field of the device's `q_info` structure. The digit in "`le0`" or "`le1`" is determined from the order in which the ILLs were opened; this index value can be overridden with the **`IF_UNITSEL`** ioctl. The IPIF index, "`:0`", "`:1`", etc., is extracted from the name supplied in the configuration ioctl which first creates the IPIF; any numeric value is acceptable. Upper level modules and applications use these names in commands to ARP and IP that affect an interface. ARP and IP both follow this naming convention.

Loopback Device

An ILL and associated IPIF for the loopback device, "`lo0`", are automatically created when the first ILL or IPC instance is opened. This is really a "pseudo-device", since no actual STREAMS device exists, nor can "`lo0`" be opened by an application. It is created solely to facilitate sending IP datagrams to the IP loopback address, 127.0.0.1.

Please note the following details of the loopback device:

- Per RFC 1122, Section 3.2.1.3, packets for any 127.*.*.* address are local and are never sent out an interface.
- The default loopback device is actually "`lo0:0`" and its address and net mask cannot be changed from the default 127.0.0.1 and 255.0.0.0, respectively. However, the interface may be brought down.
- By default, the loopback interface sets up routing entries so that packets for any 127.*.*.* address will loop back to the local host.
- Loopback interfaces "`lo0:x, x ¦ 0`", may be configured with any IP address, and that address will be treated as a loopback address.

Configuring an Interface

An interface is configured by ioctl commands from an application. The ioctls may be issued as either **`I_STR`** or transparent **`STREAMS`** ioctls. The transparent form is binary compatible with BSD socket **`SIOCxxx`** ioctls; thus BSD applications using these ioctls should function without change. New applications may prefer the **`I_STR`** interface.

The argument to the ioctl is the address of an `ifreq` structure which is idefined in **`if.h`** and shown below:

```
struct  ifreq {
#define IFNAMSIZ    36
    char    ifr_name[IFNAMSIZ];
    union {
        struct sockaddr     ifru_addr;
        struct sockaddr     ifru_dstaddr;
        struct sockaddr     ifru_broadaddr;
        short               ifru_flags;
        int                 ifru_metric;
        caddr_t             ifru_data;
    } ifr_ifru;
#define ifr_addr        ifr_ifru.ifru_addr
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr
#define ifr_flags       ifr_ifru.ifru_flags
#define ifr_metric      ifr_ifru.ifru_metric
#define ifr_data        ifr_ifru.ifru_data
};
```

Each ioctl sets or gets a single option. In all cases the *ifr_name[…]* field is a null terminated string containing the name for the interface to which the command applies. See *"ILLs and Interfaces" on page 3-67* for details about interface names. Other fields are set or returned as described below for each command.

An interface is made ready for use by issuing an **SIOCSIFFLAGS** ioctl with the **IFF_UP** bit set in the *ifr_flags* field. This action is called "bringing up" the interface. In most cases an application program, such as **ifconfig**, will convert a user request into several ioctls, and will issue these requests in an appropriate order. Wherever possible, however, IP will accept ioctls in any order. In particular, several ioctls which only make sense before an interface is marked "up", can be issued after the interface is up, and IP will take appropriate action.

IP Configuration ioctl's

The set of ioctls recognized by IP is a superset of the **SIOCxxx** requests provided in 4.3BSD. This section describes ioctls to configure an interface; routing and ARP ioctls are described in the next section (*Defining IP Routes*).

In the following paragraphs the **SIOCxxx** ioctls are paired: **SIOCSxxx** sets a value; **SIOCGxxx** retrieves the current value.

The errors listed are in addition to all the "obvious" errors. For example, all **SIOCSxxx** ioctls require privilege and return EPERM if the application does not have privilege; forgetting to set the *sa_family* field in the **sockaddr** structure to **AF_INET** or some other address specification error returns EFAULT.

IF_UNITSEL

IF_UNITSEL is used to set an interface number (PPA) for a Style 2 driver to N, without the need to create interfaces 0, 1, 2,…, N-1.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name without the numeric suffix, e.g., just “le”, not “le0”.
ifr_metric	The PPA for a **DL_ATTACH_REQ** for the driver associated with this interface, i.e., the numeric suffix to append to the interface name, e.g., the 2 in “le2”.

Errors

Error Value	Possible Cause
EBUSY	The interface is currently coming up or down.
EINVAL	This ioctl must be issued on an ILL stream over a device, not on an IPC stream.

NOTES

1. IP assumes that PPA numbers for Style 2 drivers can be allocated sequentially. starting with 0, in the order in which ILL streams are opened. **IF_UNITSEL** is used to set interface number to N, without the need to open ILLs for interfaces 0, 1, 2,…, N-1.
2. This ioctl is issued during initial configuration of IP ILL instances over a device, while the stream is still open to the configuration application and before the stream is linked under the dummy multiplexor (refer to Figure 3-2).
3. *ifr_name* must match the device name, as taken from the device’s *q_qinfo–>qi_minfo–>mi_idname* field. For more information on interface naming conventions see *“ILLs and Interfaces” on page 3-67*.

SIOCGIFADDR
SIOCSIFADDR

Set the IP address associated with the named interface.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_addr	sockaddr structure containing the IP address for this interface.

Errors

Error Value	Possible Cause
EINVAL	An attempt was made to set the address of "lo0:0".
EBADADDR	The address is not a valid local address. If the interface has address 0.0.0.0, it cannot be changed without first bringing down the interface.

NOTES

1. If no network mask has been set, a default mask is computed from the address class.
2. The broadcast address is reset to agree with the net mask.
3. If the interface is currently up, IP brings it down and then brings it back up with the new address. This action effectively clears all routing entries for the old address.

SIOCGIFCONF

This ioctl returns an array of `ifreq` structures containing the interface name and IP address for each interface currently configured. For historical reasons, the **I_STR** and transparent forms of this ioctl have different arguments.

I_STR Format

The buffer passed to the ioctl must be large enough to hold the array of `ifreq` structures that will be returned. If this buffer is not large enough to hold all the `ifreq` structures, the ioctl returns **EINVAL**.

Since there is no mechanism to return the length of the output data, the application must zero the buffer before passing it in, and then detect the end of the array of `ifreq` structures by a ioctl entry.

Transparent Format

The argument is the address of an `ifconf` structure as shown below: *ifc_req* points at a buffer of length *ifc_len* which is filled with an array of `ifreq` structures, one for each interface on the system. On output, *ifc_len* is the number of bytes in the buffer which is equal to the number of `ifreq` structures being returned times the size of an `ifreq` structure. If the buffer is not large enough to hold all the *ifreq* structures, the ioctl returns **EINVAL**.

```
struct ifconf {
    int     ifc_len;
    union {
        caddr_t         ifcu_buf;
        struct ifreq    *ifcu_req;
    } ifc_ifcu;
#define ifc_buf      ifc_ifcu.ifcu_buf
#define ifc_req      ifc_ifcu.ifcu_req
};
```

SIOCGIFFLAGS
SIOCSIFFLAGS

Bring an interface up or down.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_flags	See tables below.

Errors

Error Value	Possible Cause
EALREADY	An attempt was made to bring up an interface that is already up.

NOTES

1. **SIOCGIFFLAGS** returns a bit mask comprised of values in the following tables.
2. The following flag values may be set by **SIOCSIFFLAGS**.

Flag	Meaning
IFF_BROADCAST	Under normal usage, this flag is set by IP if the device's **DL_INFO_ACK** reports a broadcast address. This is a flag to IP that this device needs an address resolver. Some **DHCP/BOOTP** applications may need to issue broadcasts over point-to-point links, for which **IFF_BROADCAST** would normally not be set. If an application must send packets to a broadcast address over a link that does not support broadcasts, it may set this flag prior to bringing the interface up.
IFF_UP	Set this in **SIOCSIFFLAGS** to bring up an interface. Likewise, an interface is up if this is set in *ifr_flags* returned in a **SIOCGIFFLAGS** request.

3. The following flag values are controlled by IP and may be read, but not set.

Flag	Meaning
IFF_LOOPBACK	This flag is set by IP for the loopback interfaces.
IFF_MULTICAST	This flag is set if the interface supports multi-cast. IP assumes that any device which supports hardware broadcast or is a point-to-point link supports multicasts. If the **DL_ENABMULTI_REQ** fails, **IFF_MULTI_BCAST** is set and IP will use broadcasts for multicasts.
IFF_MULTI_BCAST	This flag is set when IP is using broadcasts for multicasts. See **IFF_MULTICAST**.
IFF_NOARP	This flag is set for interfaces with no resolver, which currently corresponds to all interfaces for a device without a broadcast address. This flag is set and reported by IP, but it is otherwise unused.

Flag	Meaning
IFF_POINTTOPOINT	This flag is set for point-to-point interfaces, i.e., ones for which the underlying device reported a zero-length broadcast address and zero-length physical address in the **DL_INFO_ACK** when the device was initialized.
IFF_RUNNING	This flag is always set, since it simply indicates that data structures for the interface have been created.
IFF_UNNUMBERED	Normally each IPIF is associated with a unique IP address. This flag is set for point-to-point interfaces for which no unique local address is provided (an “unnumbered” interface).

4. The following flag values are recorded by IP, but are otherwise ignored.

IFF_ALLMULTI	IFF_DEBUG	IFF_INTELLIGENT
IFF_NOTRAILERS	IFF_PRIVATE	IFF_PROMISC

5. If a network mask has not been set with the **SIOCSIFNETMASK** ioctl when an interface is brought up, a mask is created from the address class of the interface’s local address, or all 1’s for a point-to-point interface.

6. When an interface is brought up, IP checks that the interface’s address and broadcast address are in agreement. If not, it resets the broadcast address to the network broadcast based on the local address and specified (sub)network mask.

7. When the first interface on any device is brought up, a **DL_BIND_REQ** and optionally a **DL_ATTACH_REQ** are sent to the driver. Only when the **DL_BIND_ACK** or **DL_ERROR_ACK** for the bind is received will the original ioctl complete.

   When the **DL_BIND_REQ** is required, the IPIF is not marked “up” until the **T_BIND_ACK** is received and processed. This final processing includes adding all multicast addresses and, if configured to do so, sending ICMP_ADDRESS_MASK_REPLY messages for each interface on the device.

   When the **DL_BIND_REQ** is not required, i.e., other interfaces are already up, IP sends the ICMP_ADDRESS_MASK_REPLY immediately, if so configured, and returns.

SIOCGIFDSTADDR
SIOCSIFDSTADDR

Set or retrieve the IP address for a point-to-point interface.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_dstaddr	The address for the point-to-point link.

NOTES

1. Normally the interface must be a point-to-point interface, i.e., its **DL_INFO_ACK** has both a zero-length broadcast address and zero-length physical address. However, for testing purposes, **SIOCSIFDSTADDR** can be used to simulate a point-to-point link over Ethernet.
2. If not already set, **SIOCSIFDSTADDR** sets the *IFF_POINTTOPOINT* flag for the IPIF.
3. If the IPIF is currently up, IP brings it down and then brings it up with the new address; this effectively clears all routing entries for the old address.

SIOCGIFBRDADDR
SIOCSIFBRDADDR

Set or retrieve the broadcast address for an interface. The set operation is optional; IP will compute the correct broadcast address from the specified subnet mask.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_broadaddr	The broadcast address for the interface.

Errors

Error Value	Possible Cause
EBADADDR	The interface does not support a broadcast address (**IFF_BROADCAST** is not set in a **SIOCGIFFLAGS** request). If the interface is already up, the specified broadcast address must match a valid broadcast address based on the address and net mask. An **SIOCGIFBRDADDR** was issued for an interface without a broadcast address.

IP computes the broadcast address based on the address and subnet mask. By default, the broadcast address is the “all 1’s” subnet broadcast address. However there are, generally, six different broadcast addresses that are acceptable in the **SIOCSIFBRDADDR** ioctl.: “all 1's”, “all 0's”, “NET.all 1's”, “Net.all 0's”, “SubNET.all 1's”, and “SubNet.all 0's”. The address specified in this ioctl must match one of these six.

SIOCGIFMETRIC
SIOCSIFMETRIC

These ioctls are recorded but not used by IP.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_metric	The metric for the interface.

SIOCGIFMTU
SIOCSIFMTU

These ioctls set or retrieve the interface’s MTU.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_metric	The MTU for the interface.

Errors

Error Value	Possible Cause
EINVAL	The specified MTU is less than 72 bytes.

NOTES

1. All routing entries using the specified interface will use the specified MTU.
2. The value specified may be larger than the *dl_max_sdu* value reported by the driver. This feature may be necessary for some drivers which report an artificially small *dl_max_sdu*.

SIOCSIFNETMASK
SIOCSIFNETMASK

These ioctls set or retrieve the interface’s subnet mask.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.
ifr_addr	The subnet mask for the interface.

Errors

Error Value	Possible Cause
`EBADADDR`	An attempt was made to change the subnet mask for “lo0:0” loopback device. The subnet mask is improperly formed or is inconsistent with the IP address for the interface. An **`SIOCGBRDADDR`** was issued for an interface without a broadcast address.

If the interface is currently up, setting the subnet mask will cause IP to bring the interface down and back up with the new subnet mask. All routing entries associated with the previous address and subnet mask will be deleted.

SIOCGIFSTATS

Return an `ifrecord` structure corresponding to the specified interface.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., “le0”.

SIOCGIFSTATS_ALL

This ioctl returns an array of `ifrecord` structures containing the interface name and statistics for each interface currently configured. This ioctl uses the **`I_STR`** format. The buffer passed to the ioctl must be large enough to hold the array of `ifrecord` structures that will be returned. If this buffer is not large enough to hold all the `ifreq` structures, the ioctl returns as many structures as can be held in the buffer.

Since there is no mechanism to return the length of the output data, the application must zero the buffer before passing it in, and then detect the end of the array of `ifreq` structures by a ioctl entry.

SIOCGETSGCNT

Returns the packet, byte, and RPF failure counts for the specified source-group. The ioctl argument is an `sioc_sg_req` structure defined in **`ip_mroute.h`** and shown below:

```
struct sioc_sg_req {
    struct in_addr      src;
    struct in_addr      grp;
    ulong               pktcnt;
    ulong               bytecnt;
    ulong               wrong_if;
};
```

The *src* and *grp* fields must be filled in with the source IP and group multicast addresses for which statistics are wanted. The other three fields are returned.

SIOCGETVIFCNT

Returns the packet and byte counts for the specified VIF. The ioctl argument is an `sioc_vif_req` structure defined in **`ip_mroute.h`** and shown below:

```
struct sioc_vif_req {
    vifi_t  vifi;   /* vif number */
    ulong   icount;/* Input packet count on vif */
    ulong   ocount;/* Output packet count on vif */
    ulong   ibytes;/* Input byte count on vif */
    ulong   obytes;/* Output byte count on vif */
};
```

The *vifi* field must be filled in with the VIF index for which statistics are wanted. The other four fields are returned.

SIOCGIFNUM

Returns the number IPIFs across all ILLs.

Arguments

Field	Contents
ifr_name	NULL-terminated interface name, e.g., "le0".
ifr_metric	The count of IPIFs.

Defining IP Routes

This section describes ioctls which are used to manipulate IP's routing table. ioctls processed by IP for managing the ARP cache are also described.

The argument to the routing ioctls is a pointer to an `rtentry` structure, defined in **`route.h`** and shown below:

```
struct rtentry {
    u_long              rt_hash;
    struct sockaddr     rt_dst;
    struct sockaddr     rt_gateway;
    short               rt_flags;
    short               rt_refcnt;
    ulong               rt_use;
    struct ip_provider  *rt_prov;
    int                 rt_metric;
    int                 rt_proto;
    time_t              rt_age;
};
```

The *rt_dst*, *rt_gateway*, and *rt_flags* fields are used for specifying routes. The *rt_hash*, *rt_refcnt* and *rt_use* fields are unused.

SIOCADDRT
SIOCDELRT

The **SIOCADDRT** ioctl is used to add routes to hosts, networks, and to define gateways; **SIOCDELRT** deletes routes. The argument to the ioctl is a pointer to an *rtentry* structure. The fields used in the *rtentry* structure depend upon the type of route entry being created.

The following cases occur:

1. Adding or deleting a gateway to use for a specific host address. This entry has precedence over a route for a subnet to which the specified host address belongs

Field	Value
rt_gateway	The IP address of the gateway.
rt_dst	The IP address of the remote host.
rt_flags	**RTF_GATEWAY** set. **RTF_HOST** set.

2. Adding or deleting a gateway to use for a network address

Field	Value
rt_gateway	The IP address of the gateway.
rt_dst	The network address.
rt_flags	**RTF_GATEWAY** set. **RTF_HOST** clear.

3. Adding or deleting a default gateway. Multiple default gateways may be specified; they will be used in round-robin order

Field	Value
rt_gateway	The IP address of the gateway.
rt_dst	0.0.0.0
rt_flags	**`RTF_GATEWAY`** set. **`RTF_HOST`** clear (bit ignored in this case).

4. Adding or deleting the interface to use for a directly connected subnet.

Field	Value
rt_gateway	The local IP address of the interface to use.
rt_dst	The subnet address.
rt_flags	**`RTF_GATEWAY`** clear. **`RTF_HOST`** clear.

5. Adding or deleting the interface to use for a directly connected host

Field	Usage
rt_gateway	The local IP address of the interface to use.
rt_dst	The IP address of the remote host.
rt_flags	**`RTF_GATEWAY`** clear. **`RTF_HOST`** set.

NOTES

1. If **`RTF_HOST`** is clear (cases 2 and 4), the subnet mask, as specified or as deduced from the address class, and the address in *rt_dst* must satisfy *(mask & rt_dst)* == *rt_ds*t.

2. 2. BSD-style proxy ARP Support.
   IP supports **`route`** commands of the form:

   ```
   route add host <host address> <our address> 0
   route add net <net address> <our address> 0
   ```

   where **`<host address>`** and **`<net address>`** are not included in any current directly connected subnet, and **`<our address>`** is an address assigned to some local interface. Additionally, **`<net address>`** or **`<host address>`** may also be 0.0.0.0. These **`route`** commands are implemented using **`SIOCADDRT`** ioctls of form 4 and 5 above.

   Once such an entry has been added IP expects that any destination which matches **`<net address>`** or **`<host address>`** will be ARP'ed for out the interface associated with **`<our address>`**.

When **`<net address>`** or **`<host address>`** is equal to `0.0.0.0`, this entry functions like a default gateway except that it does not reside in the default gateway list and is not subject to dead gateway detection. Since this entry does not reside in the default gateway list it is also not subject to the round robin selection process used for selection of normal default gateways. In addition, entries of this form will take precedence over any existing default gateways.

In general these entries should only be used when a host knows that some host or router will answer to ARP's for a destination address which is not directly connected and for which the host does not have a more specific host or net route to the destination.

3. If a configuration ioctl causes an interface to be brought down and back up, all existing routes using that interface are deleted, and any **`SIOCADDRT`** commands must be re-issued. Note, however, that routing entries created for case 5, are not removed.

Errors

The numbers in parentheses refer to the cases above.

Error Code	Possible Cause(s)
`ENETUNREACH`	(1 through 3) The address in *rt_gateway* is not on a directly connected subnet. (4 and 5) No interface is associated with the local address specified in *rt_gateway.*
`EHOSTUNREACH`	(5) The local address in *rt_gateway* corresponds to a point-to-point link, but *rt_dst* does not match destination IP address for this link.
`EEXIST`	(1 through 5) The IP address in *rt_dst* matches an address for some interface on this host. (1 through 3) An entry for this gateway already exists. (5) There is already a route for the specified host.

SIOCFLUSHRT

This ioctl is used to delete all routes which have been added by **`SIOCADDRT`**, plus all routes learned through ARP responses. The argument to this ioctl is unused and may be NULL.

ARP ioctls

IP supports three “Berkeley ARP ioctls” for manipulating the ARP cache. IP converts these ioctls into commands which are passed to ARP for processing; ARP’s response is returned to IP, which in turn completes the ioctl. An alternate ARP command interface is described in Chapter 1.

The ARP ioctls pass an argument that is a pointer to an `arpreq` structure defined in *i*`if_arp.h` and shown below:

```
struct arpreq {
    struct sockaddr     arp_pa;
    struct sockaddr     arp_ha;
    int                 arp_flags;
};
/*  arp_flags and at_flags field values */
#define     ATF_INUSE           0x01
#define     ATF_COM             0x02
#define     ATF_PERM            0x04
#define     ATF_PUBL            0x08
#define     ATF_USETRAILERS     0x10
```

arp_pa is the protocol address, *arp_ha* is the hardware address.

SIOCDARP

Remove an entry from the ARP cache.

Arguments

Field	Usage
arp_flags	Unused. Must be zero.
arp_pa	The IP address of the entry to delete.
arp_ha	Unused. Must be zero.

SIOCGARP

Retrieve information about an ARP cache entry.

Arguments

Field	Usage
arp_flags	Returned mask has one or more of the following set: **`ATF_INUSE`** - set if IP has a routing table entry for *arp_pa*. **`ATF_COM`** - set if routing table entry has a hardware address for given IP address. **`ATF_LOCAL`** - set if entry is for a local IP address. **`ATF_PERM`** - set if entry will not be deleted by periodic timer-initiated ARP cache flushing. **`ATF_PUBL`** - set if this entry will be used by ARP to respond to ARP requests from remote hosts. **`ATF_USETRAILERS`** - always clear; unused by TCP/IP.
arp_pa	The IP address of the entry for which information is sought.
arp_ha	The hardware address and length from the ARP cache entry are returned.

SIOCSARP

Add an entry to the ARP cache. An entry may be for a single IP address or an entire network or subnet range of IP addresses.

Arguments

Field	Usage
arp_flags	Set **`ATF_PERM`** to create a permanent entry (will only be removed by an **`SIOCDARP`** ioctl. Set **`ATF_PUBL`** if this entry can be used to respond to ARP requests from remote hosts.
arp_pa	The IP address for this entry.
arp_ha	The hardware address is stored in *arp_ha.sa_data*, spilling over into *arp_ha_pad* if necessary. *arp_ha.sa_family* is unused.

Index

A

Accepting connect indications 2-5
ACK timer thresholds (TCP) 1-11
ARP
  IPioctls 3-18
arpreq structure 3-18

B

Big endian 2-4, 2-9, 2-23
Binding to a Protocol
  RAWIP 2-24
Binding to Addresses and Ports Already in Use 2-14
Binding to addresses in use
  TCP 2-4
Broadcast Binds 2-15
BSD
  proxy ARP support 3-16
  socket ioctls 3-4
Byte order 2-4, 2-9, 2-23

C

Configuring an interface 3-4
Connected UDP 2-16
Copy avoidance
  TCP option 2-13

D

DATA_REQ 2-6
Dead gateways
  probe interval 1-4
  with BSD proxy ARP 3-17
Default gateways 3-16
Deferred ACKs (TCP) 1-11
DHCP 2-18
Distribute inbound UDP datagrams 2-21
Dynamic Host Configuration Protocol (DHCP) 2-18

E

Errors
  EBADADDR 3-5
DATA_IND and T_ 2-6
T_ 2-6

F

Flushing data
  T_DISCON_IND 2-7
  T_DISCON_REQ 2-7

G

Gateway
  host routes 3-15
  network routes 3-15
Gateways
  dead gateway detection 1-4
  default 3-16

H

Header files
  if.h 3-4
  if_arp.h 3-18
  ip_mroute.h 3-14
  route.h 3-15
  timod.h 2-3, 2-14, 2-24
Host routes 3-15

I

I_FDINSERT ioctl 2-5
ICMP
  after UDP disconnect 2-16

  passing messages upstream 1-13
  receiving ICMP messages 2-20
  sending ICMP_ADDRESS_MASK_REPLY 3-10
  T_UDERROR_IND 2-16
if.h 3-4
if_arp.h 3-18
IF_INDEXSEL
  IP ioctl 3-5
IF_UNITSEL
  IP ioctl 3-5
ifconf structure 3-7
ifconfig program 3-5
IFF_ POINTTOPOINT 3-11
IFF_BROADCAST 3-9
IFF_LOOPBACK 3-9
IFF_MULTI_BCAST 3-9
IFF_MULTICAST 3-9
IFF_NOARP 3-9
IFF_POINTTOPOINT 3-10
IFF_RUNNING 3-10
IFF_UNNUMBERED 3-10
IFF_UP 3-5, 3-9
ifreq structure 3-4
ILL (IP Link Level) 3-1
INADDR_ANY
  bound state 2-3
  loopback usage 2-16
Interfaces
  configuration ioctls 3-4
  DL_ATTACH_REQ 3-10
  DL_BIND_REQ 3-10
  names 3-3
  net mask 3-12
  point-to-point 3-1
  PPA in IP 3-6
  setting MTU 3-12
  subnet mask 3-12
IP
  adding routes 3-15
  defining routes 3-14
  interface names 3-3
  tunable parameters 1-3
IP Client (IPC) 3-1
IP ioctls
  IF_INDEXSEL 3-5
  IF_UNITSEL 3-5
  SIOCADDRT 3-15
  SIOCDARP 3-18
  SIOCDELRT 3-15
  SIOCFLUSHRT 3-17
  SIOCGARP 3-18
  SIOCGETSGCNT 3-14
  SIOCGETVIFCNT 3-14
  SIOCGIFADDR 3-6
  SIOCGIFBRDADDR 3-11
  SIOCGIFCONF 3-7
  SIOCGIFDSTADDR 3-10
  SIOCGIFFLAGS 3-8
  SIOCGIFMETRIC 3-12
  SIOCGIFMTU 3-12
  SIOCGIFNUM 3-14
  SIOCGRTENTRY 3-18
  SIOCSARP 3-19
  SIOCSIFADDR 1-3, 3-6
  SIOCSIFDSTADDR 3-10
  SIOCSIFFLAGS 3-5, 3-8
  SIOCSIFMETRIC 3-12
  SIOCSIFMTU 3-12
  SIOCSIFNETMASK 3-12
  SIOCSIFSWITCH 3-14
IP_ADD_MEMBERSHIP 2-22
IP_BROADCAST
  RAWIP 2-27
  TCP 2-9
  UDP 2-18
IP_BROADCAST_IF 2-18
IP_BROADCAST_IFNAME 2-18
IP_DISTRIBUTE UDP option 2-21
IP_DONTROUTE
  RAWIP 2-27
  TCP 2-9
  UDP 2-19
IP_DROP_MEMBERSHIP 2-22
IP_HDRINCL 2-26, 2-28
ip_mreq 2-22
ip_mreq structure 2-22
ip_mroute.h 3-14
IP_MULTICAST_IF 2-23
IP_MULTICAST_LOOP 2-23
IP_MULTICAST_TTL 2-23
IP_OPTIONS
  RAWIP 2-28
  TCP 2-9
  UDP 2-19
IP_RECVDSTADDR 2-18 to 2-19
IP_RECVIFADDR 2-18 to 2-19
IP_RECVOPTS 2-18, 2-20
IP_REUSEADDR
  TCP 2-4, 2-9
  UDP 2-14, 2-20
IP_REUSEPORT
  TCP 2-4, 2-9
  UDP 2-14, 2-20
IP_TOS
  RAWIP 2-28
  TCP 2-9
  UDP 2-20
IP_TTL
  RAWIP 2-28
  TCP 2-9
  UDP 2-20
ip_xxx tunable parameters 1-3

IPC 3-1
IPIF (definition) 3-3
IPPROTO_ICMP with RAWIP 2-23
IPPROTO_RAW with RAWIP 2-26

K

Keep-alive strategies 2-10
Keep-alives in TCP 1-12
“Killer” keep-alives 2-10

L

linger structure 2-12
Listener streams 2-5
Loopback address
  sending to INADDR_ANY 2-16
Loopback device
  ILL and IPIF creation 3-4

M

mfcctl structure 2-30
MGMT_flags Field in T_optmgmt_req 2-31
MORE_flag 2-6
MRT_ADD_MFC 2-30
MRT_ADD_VIF 2-30
MRT_ASSERT 2-29
MRT_DEL_MFC 2-30
MRT_DEL_VIF 2-30
MRT_DONE 2-30
MRT_INIT 2-30
MRT_VERSION 2-31
MSS 2-12
MTU
  path MTU discovery 1-6
  SIOCSIFMTU ioctl 3-12
Multicast
  interface selection 2-23
  local loopback 2-23
  time-to-live 2-23
  UDP options 2-22
Multicast routing
  UDP options 2-29

N

Nagle limit 1-9, 2-11
Named dispatch 1-1 to 1-2
Names of interfaces 3-3
Network byte order 2-4, 2-9, 2-23
Network routes 3-15
Note on Connection Establishment 2-6

O

opthdr structure 2-32
Options
  on T_UNITDATA_IND 2-20
  with T_UNITDATA_IND 2-17
  with TCP T_CONN_REQ 2-6
Options Management 2-31
Options management
  TLI semantics 2-32
Options with T_UNITDATA_REQ 2-17
Other TPI-Related Issues 2-8

P

Path MTU discovery 1-6
Point-to-point interfaces 3-1
PPA 3-5
Proxy ARP Support 3-16

Q

q_maxpsz 2-8
q_minpsz 2-8

R

RAWIP
  binding to a Protocol 2-24
  tunable parameters 1-7
RAWIP option
  IP_BROADCAST 2-27
  IP_DONTROUTE 2-27
  IP_HDRINCL 2-26, 2-28
  IP_OPTIONS 2-28
  IP_TOS 2-28
  IP_TTL 2-28
  XTI_DEBUG 2-28
  XTI_PROTOTYPE 2-29
  XTI_RCVBUF 2-29
  XTI_RCVLOWAT 2-29

XTI_SNDBUF 2-29
XTI_SNDLOWAT 2-29
RAWIP Options 2-27
RAWIP Options for IP Multicast 2-31
RAWIP Options for Multicast Routing 2-29
rawip_bsd_compat 2-27
rawip_xxx tunable parameters 1-7
RFC 1122
Sec 3.2.1.3 1-3, 3-4
Sec 4.2.2.21 1-8
Sec 4.2.2.4 1-9
Sec 4.2.3.2 1-11
Sec 4.2.3.4 1-9, 2-11
Sec 4.2.3.5 1-11, 2-10
RFC 1191 1-6
RFC 793 1-9, 2-1
Round trip time-out 1-11
route.h 3-15
Routing 3-14
host routes 3-15
network routes 3-15
rtentry structure 3-15
RTF_HOST flag 3-16

S

Semantics of T_OPTMGMT_REQ Message Formats 2-31
sioc_sg_req structure 3-14
sioc_vif_req structure 3-14
SIOCADDRT 3-15
SIOCDARP 3-18
SIOCDELRT 3-15
SIOCFLUSHRT 3-17
SIOCGARP 3-18
SIOCGETSGCNT 3-14
SIOCGETVIFCNT 3-14
SIOCGIFADDR 3-6
SIOCGIFBRDADDR 3-11
SIOCGIFCONF 3-7
SIOCGIFDSTADDR 3-10
SIOCGIFFLAGS 3-8
SIOCGIFMETRIC 3-12
SIOCGIFMTU 3-12
SIOCGIFNUM 3-14
SIOCGRTENTRY 3-18
SIOCSARP 3-19
SIOCSIFADDR 1-3, 3-6
SIOCSIFBRDADDR 3-11
SIOCSIFDSTADDR 3-10
SIOCSIFFLAGS 3-5, 3-8
SIOCSIFMETRIC 3-12
SIOCSIFMTU 3-12
SIOCSIFNETMASK 3-12
SIOCSIFSWITCH 3-14
SNMP
deleteTCB for TCP 1-12
SO_KEEPALIVE socket option 2-35
SO_TYPE socket option 2-35
SO_USELOOPBACK socket option 2-35
sockaddr_in structure 2-3
Socket ioctls
SIOCADDRT 3-15
Socket option
SO_KEEPALIVE 2-35
SO_KEEPALIVE in TCP 1-12
SO_TYPE 2-35
SO_USELOOPBACK 2-35
Socket states 2-3
SYN attack defense 1-9

T

T_ADDR_REQ and T_ADDR_ACK 2-3, 2-13
T_BIND_RE “Re-bind” Request Operation 2-5
T_BIND_REQ
as ioctl 2-14
UDP multicast usage 2-14
T_BIND_REQ “Address in use” Errors 2-5, 2-15
T_BIND_REQ and T_BIND_ACK 2-3, 2-14, 2-24
T_BIND_REQ as ioctl 2-3
T_CHECK 2-31
T_CONN_IND and T_CONN_RES 2-5
T_CONN_REQ and T_CONN_CON 2-6, 2-16, 2-25
T_CURRENT 2-9, 2-32
T_DEFAULT 2-32
T_DISCON_IND
during accept 2-6
flushing data 2-7
T_DISCON_IND and T_DISCON_REQ 2-7
T_DISCON_REQ 2-16, 2-25
T_DISCON_REQ
flushing data 2-7
rejecting connect indications 2-7
T_INFO_REQ and T_INFO_ACK 2-7, 2-16, 2-26
T_NEGOTIATE 2-32
T_ORDREL_REQ and T_ORDREL_IND 2-8
T_UDERROR_IND
connected UDP 2-16
ICMP messages 2-20
T_UNBIND_REQ 2-8, 2-17, 2-26
T_UNITDATA_IND 2-27
T_UNITDATA_IND options 2-17
T_UNITDATA_REQ and M_DATA 2-26
T_UNITDATA_REQ and T_UNITDATA_IND 2-17
T_UNITDATA_REQ options 2-17
TCP
ACK timer thresholds 1-11
deferred ACKs 1-11
forcing connection termination 1-12

keep-alives 1-12
MSS 2-12
MSS and q_maxpsz 2-8
PSH bit 1-10
receive window 2-12
round trip time-out 1-11
state machine 2-1
SYN-ACK handshake 2-6
TIME_WAIT state 2-6, 2-8
TPI states 2-3
tunable parameters 1-7
urgent data pointer 2-11
TCP option
IP_BROADCAST 2-9
IP_DONTROUTE 2-9
IP_OPTIONS 2-9
IP_REUSEADDR 2-9
IP_REUSEPORT 2-9
IP_TOS 2-9
IP_TTL 2-9
TCP_ABORT_THRESHOLD 2-10
TCP_CONN_ABORT_THRESHOLD 2-10
TCP_CONN_NOTIFY_THRESHOLD 2-10
TCP_KEEPALIVE 2-10
TCP_MAXSEG 2-11
TCP_NODELAY 2-11
TCP_NOTIFY_THRESHOLD 2-10
TCP_OOBINLINE 2-11
TCP_URGENT_PTR_TYPE 2-11
XTI_DEBUG 2-11
XTI_LINGER 2-12
XTI_PROTOTYPE 2-12
XTI_RCVBUF 2-12
XTI_SND_COPYAVOID 2-13
XTI_SNDBUF 2-13
XTI_SNDLOWAT 2-13
TCP Options 2-8
TCP State Machine 2-1
TCP_ABORT_THRESHOLD 2-10
TCP_CONN_ABORT_THRESHOLD 2-10
TCP_CONN_NOTIFY_THRESHOLD 2-10
tcp_def_ttl 2-9
tcp_ip_abort_cinterval 2-10
tcp_ip_notify_cinterval 2-10
tcp_ip_notify_interval 2-10
TCP_KEEPALIVE 2-10
tcp_keepalive_interval 2-35
tcp_killer_keepalives 1-12, 2-11
TCP_MAXSEG 2-11
TCP_NODELAY 2-11
TCP_NOTIFY_THRESHOLD 2-10
TCP_OOBINLINE 2-11
tcp_rcv_push_wait 2-12
tcp_recv_hiwater_def 2-12
TCP_URGENT_PTR_TYPE 2-11
tcp_xxx tunable parameters 1-7
tcpConnState
MIB-II details 1-13
TI_BIND
T_BIND_REQ as ioctl 2-14
TI_BIND ioctl
T_BIND_REQ
as ioctl 2-3
TIME_WAIT state 2-6, 2-8
timod.h 2-3, 2-14, 2-24
TLI Semantics for Options Processing 2-32
TPI
MORE_flag 2-6
TPI Primitives for RAWIP 2-23
TPI Primitives for TCP 2-1
TPI Primitives for UDP 2-13
TPI states 2-3
Tunable parameters
how to set 1-2
IP 1-3
RAWIP 1-7
TCP 1-7
UDP 1-13

U

UDP
broadcast binds 2-15
connected stream 2-16
ICMP messages upstream 1-13
message boundaries 2-16
receiving ICMP messages 2-20
tunable parameters 1-13
UDP option
IP_ADD_MEMBERSHIP 2-22
IP_BROADCAST 2-18
IP_BROADCAST_IF 2-18
IP_BROADCAST_IFNAME 2-18
IP_DISTRIBUTE 2-21
IP_DONTROUTE 2-19
IP_DROP_MEMBERSHIP 2-22
IP_MULTICAST_IF 2-23
IP_MULTICAST_LOOP 2-23
IP_MULTICAST_TTL 2-23
IP_OPTIONS 2-19
IP_RECVDSTADDR 2-18 to 2-19
IP_RECVIFADDR 2-18 to 2-19
IP_RECVOPTS 2-18, 2-20
IP_REUSEADDR 2-14, 2-20
IP_REUSEPORT 2-14, 2-20
IP_TOS 2-20
IP_TTL 2-20
MRT_ADD_MFC 2-30

MRT_ADD_VIF 2-30
MRT_ASSERT 2-29
MRT_DEL_MFC 2-30
MRT_DEL_VIF 2-30
MRT_DONE 2-30
MRT_INIT 2-30
MRT_VERSION 2-31
multicast 2-22
multicast routing 2-29
UDP_CHECKSUM 2-20
UDP_RX_ICMP 2-16, 2-20
XTI_DEBUG 2-21
XTI_PROTOTYPE 2-21
XTI_RCVBUF 2-21
XTI_RCVLOWAT 2-21
XTI_SNDBUF 2-21
XTI_SNDLOWAT 2-22
UDP Options 2-18
UDP Options for IP Multicast 2-22
UDP_CHECKSUM 2-20
udp_def_ttl 2-20
udp_pass_up_icmp 2-16, 2-20
UDP_RX_ICMP 2-16, 2-20
UDP_RX_ICMP<default parafont>
UDP 2-16
udp_xxx tunable parameters 1-13
Urgent data pointer 2-11

V

vifctl structure 2-30

X

XPG4_1
with UDP 2-13
XPG4_1 with TCP 2-3, 2-7
XTI
states 2-3
XTI_DEBUG
RAWIP 2-28
TCP 2-11
UDP 2-21
XTI_LINGER 2-12
XTI_PROTOTYPE
RAWIP 2-29
TCP 2-12
UDP 2-21
XTI_RCVBUF
RAWIP 2-29
TCP 2-12
UDP 2-21
XTI_RCVLOWAT
RAWIP 2-29
UDP 2-21
XTI_SND_COPYAVOID
TCP 2-13
XTI_SNDBUF
RAWIP 2-29
TCP 2-13
UDP 2-21
XTI_SNDLOWAT
RAWIP 2-29
TCP 2-13
UDP 2-22