ChorusOS man pages section 7P: Protocols

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3339
December 10, 1999

Contents

PREFACE 5

Intro(7P) 11

arp(7P) 13

icmp(7P) 15

inet(7P) 16

ip(7P) 20

route(7P) 25

tcp(7P) 29

udp(7P) 32

Index 33

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME — This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS — This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] — The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . — Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .

| — Separator. Only one of the arguments separated by this character can be specified at time.

{ } — Braces. The options and / or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

Protocols

10

NAME

Intro – introduction to networking facilities

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
```

DESCRIPTION

This section briefly describes the networking facilities available on the system.

All network protocols are associated with a specific "protocol family". A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing and basic transport. A protocol family may support multiple methods of addressing, even if the current protocol implementations do not. A protocol family is normally comprised of a number of protocols, one per `socket`(2POSIX) type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in `socket`(2POSIX). A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family / network architecture. Certain semantics of the basic socket abstractions are protocol—specific. All protocols should support the basic model for their particular socket type, but may also provide non-standard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction may allow more than one byte of out-of-band data to be transmitted per out-of-band message.

A network interface is similar to a device interface. Network interfaces comprise the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families and / or address formats.

PROTOCOLS

The system currently supports the DARPA Internet protocols. Raw socket interfaces are provided to the IP protocol layer of the DARPA Internet. Consult the appropriate manual pages in this section for more information regarding the support for each protocol family.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`socket`(2POSIX), `ioctl`(2POSIX)

Name	Description
arp(7P)	Address Resolution Protocol
icmp(7P)	Internet Control Message Protocol
inet(7P)	Internet protocol family
ip(7P)	Internet Protocol
route(7P)	kernel packet forwarding database
tcp(7P)	Internet Transmission Control Protocol
udp(7P)	Internet User Datagram Protocol

NAME

arp – Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

The Address Resolution Protocol (ARP) is a protocol used to map dynamically between Internet host addresses and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers. It is not specific to Internet protocols or to 10Mb/s Ethernet, but the current implementation only supports that combination.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a response to a mapping request; only the most recently "transmitted" packet is kept. If the target host does not respond after several requests, the host is considered to be down for a short period (normally 20 seconds), allowing an error to be returned to transmission attempts during this interval. The error is `EHOSTDOWN` for a non-responding destination host, and `EHOSTUNREACH` for a non-responding router.

The ARP cache is stored in the system routing table as dynamically-created host routes. The route to a directly-attached Ethernet network is installed as a *cloning* route (one with the `RTF_CLONING` flag set). This causes routes to individual hosts on that network to be created on demand. These routes time out periodically (normally 20 minutes after validation; entries are not validated when not in use). An entry for a host which is not responding is a *reject* route (one with the *RTF_REJECT* flag set).

ARP entries may be added, deleted or changed using the `arp`(1M) utility. Entries added manually may be temporary or permanent, and may be *published*. Entries which are *published* cause the system to respond to ARP requests for that host as if it were the target of the request.

In the past, ARP was used to negotiate the use of a trailer encapsulation. This is no longer supported.

ARP watches passively for hosts impersonating the local host (in other words, a host which responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

The error message: "duplicate IP address %x!! sent from ethernet address: %x:%x:%x:%x:%x:%x." indicates that ARP has discovered another host on the local network which responds to mapping requests for its own Internet address with a different Ethernet address, generally indicating that two hosts are attempting to use the same Internet address.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`arp`(1M), `ifconfig`(1M), `route`(1M), `inet`(7P), `route`(7P)

An Ethernet Address Resolution Protocol, Plummer D., RFC826

Trailer Encapsulations, Leffler S.J., Karels M.J., RFC893

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

int **socket**(*AF_INET*, *SOCK_RAW*, *proto*);

DESCRIPTION

ICMP is the error and control message protocol used by IP and the Internet protocol family. It may be accessed through a “raw socket” for network monitoring and diagnostic functions. Refer to *getsockopt*(2POSIX) for the *proto* parameter to the socket call to create an ICMP socket. ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls. The *connect*(2POSIX) call may also be used to fix the destination for future packets (in which case the `read`(2POSIX) or *recv*(2POSIX) and `write`(2POSIX) or *send*(2POSIX) system calls may be used).

Outgoing packets automatically have an IP header prepended to them (based on the destination address). Incoming packets are received with the IP header and options intact.

DIAGNOSTICS

A socket operation may fail and return one of the following error conditions:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected.
[ENOTCONN]	when trying to send a datagram with no destination address specified, and the socket is not connected.
[ENOBUFS]	when the system runs out of memory for an internal data structure.
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`send`(2POSIX), `recv`(2POSIX), `intro`(7P), `inet`(7P), `ip`(7P)

NAME

inet – Internet protocol family

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
```

DESCRIPTION

The Internet protocol family is a collection of protocols layered on top of the *Internet Protocol* (IP) transport layer, using the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

ADDRESSING

Internet addresses are four byte quantities, stored in network standard format (on i386 based machines, these are word and byte reversed). The include file <*netinet/in.h*> defines this address as a discriminated union.

Sockets bound to the Internet protocol family use the following addressing structure,

```
struct sockaddr_in {
   u_char       sin_len;
   u_char       sin_family;
   u_short      sin_port;
   struct       in_addr sin_addr;
   char         sin_zero[8];
};
```

Sockets can be created with a INADDR_ANY local alddress to enable “wildcard” matching on incoming messages. The address in a *connect*(2POSIX) or *sendto*(2POSIX) call may be given as INADDR_ANY to mean “this host.” The distinguished address INADDR_BROADCAST is allowed as shorthand for the broadcast address on the primary network (if the first network configured supports broadcast).

PROTOCOLS

The Internet protocol family is comprised of the IP transport protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction, while UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of the type SOCK_RAW. The ICMP message protocol is accessible from a raw socket.

The 32-bit Internet address contains both network and host parts. However, direct examination of addresses is discouraged. For those programs which must absolutely break addresses into their component parts, the following *ioctl*(2POSIX) commands on a datagram socket in the Internet domain.

SIOCSIFNETMASK Set interface network mask. The network mask defines the network part of the address; if it

	contains more of the address than the address type would indicate, subnets are in use.
SIOCGIFNETMASK	Get interface network mask.

ROUTING

The current implementation of Internet protocols includes some routing-table adaptations to provide enhanced caching of certain end-to-end information necessary for Transaction TCP and Path MTU Discovery. The following changes are the most significant:

1. All IP routes, except those with the RTF_CLONING flag and those to multicast destinations, have the RTF_PRCLONING flag forcibly enabled (they are thus said to be "protocol cloning").
2. When the last reference to an IP route is dropped, the route is examined to determine if it was created by cloning such a route. If this is the case, the RTF_PROTO3 flag is turned on, and the expiration timer is initialized to go off in `net.inet.ip.rtexpire` seconds.

   If such a route is re-referenced, the flag and expiration timer are reset.
3. A kernel timeout runs once every ten minutes, or sooner if there are soon-to-expire routes in the kernel routing table, and deletes the expired routes.

A dynamic process is in place to modify the value of `net.inet.ip.rtexpire` if the number of cached routes grows too large. If after an expiration run there are still more than `net.inet.ip.rtmaxcache` unreferenced routes remaining, the rtexpire value is multiplied by 3/4, and any routes which have longer expiration times have those times adjusted. This process is damped somewhat by specification of a minimum rtexpire value (`net.inet.ip.rtminexpire`), and by restricting the reduction to once in a ten-minute period.

If some external process deletes the original route from which a protocol-cloned route was generated, the "child route" is deleted. (This is actually a generic mechanism in the routing code support for protocol-requested cloning.)

No attempt is made to manage routes which were not created by protocol cloning; these are assumed to be static, under the management of an external routing process, or under the management of a link layer (for example, ARP for Ethernets).

Only certain types of network activity will result in the cloning of a route using this mechanism. Specifically, those protocols (such as TCP and UDP) which themselves cache a long-lasting reference to route for a destination will trigger the mechanism; whereas raw IP packets, whether locally-generated or forwarded, will not.

MIB VARIABLES

A number of variables are implemented in the `net.inet` branch of the `sysctl`(3POSIX) MIB. In addition to the variables supported by the transport protocols (for which see the respective manual pages), the following general variables are defined:

IPCTL_FORWARDING (ip.forwarding)

Boolean: enable / disable forwarding of IP packets (default false).

IPCTL_SENDREDIRECTS (ip.redirect)

Boolean: enable / disable sending of ICMP redirects in response to unforwardable IP packets (default true).

IPCTL_DEFTTL (ip.ttl)

Integer: default time-to-live ("TTL") to use for outgoing IP packets (default 64).

IPCTL_SOURCEROUTE (ip.sourceroute)

Boolean: enable / disable forwarding of source-routed IP packets (default false).

IPCTL_RTEXPIRE (ip.rtexpire)

Integer: lifetime in seconds of protocol-cloned IP routes after the last reference drops (default one hour). This value varies dynamically as described above.

IPCTL_RTMINEXPIRE (ip.rtminexpire)

Integer: minimum value of `ip.rtexpire` (default ten seconds). This value has no effect on user modifications, but restricts the dynamic adaptation described above.

IPCTL_RTMAXCACHE (ip.rtmaxcache)

Integer: trigger level of cached, unreferenced, protocol-cloned routes which initiates dynamic adaptation (default 128).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `ioctl`(2POSIX), `socket`(2POSIX), `sysctl`(3POSIX), `intro`(7P), `tcp`(7P), `udp`(7P), `ip`(7P), `icmp`(7P)

WARNING The Internet protocol support is subject to change as the Internet protocols develop. Users should not rely on details of the current implementation, but rather on the services exported.

NAME

ip – Internet Protocol

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int **socket**(*AF_INET*, *SOCK_RAW*, *proto*);

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a “raw socket” when developing new protocols, or special-purpose applications.

There are several IP-level setsockopt(2POSIX) and getsockopt(2POSIX) options.

IP_OPTIONS may be used to provide IP options to be transmitted in the IP header of each outgoing packet or to examine the header options on incoming packets. IP options may be used with any socket type in the Internet family. The format of IP options to be sent is that specified by the IP protocol specification (RFC-791), with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. To disable previously specified options, use a zero-length buffer:

setsockopt(*s*, *IPPROTO_IP*, *IP_OPTIONS*, *NULL*, *0*);

IP_TOS and IP_TTL may be used to set the type-of-service and time-to-live fields in the IP header for SOCK_STREAM and SOCK_DGRAM sockets. For example,

```
int tos = IPTOS_LOWDELAY;       /* see <netinet/in.h> */
setsockopt(s, IPPROTO_IP, IP_TOS, &tos, sizeof(tos));

int ttl = 60;                   /* max = 255 */
setsockopt(s, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
```

If the IP_RECVDSTADDR option is enabled on a SOCK_DGRAM socket, the recvmsg(2POSIX) call will return the destination IP address for a UDP data-gram. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by the IP address. The cmsghdr fields have the following values:

```
cmsg_len = sizeof(struct in_addr)
cmsg_level = IPPROTO_IP
cmsg_type = IP_RECVDSTADDR
```

IP_PORTRANGE may be used to set the port range used for selecting a local port number on a socket with an unspecified (zero) port number. It has the following possible values:

IP_PORTRANGE_DEFAULT use the default range of values, normally IPPORT_RESERVED

	through IPPORT_RESERVED. This is adjustable through the sysctl(3POSIX) setting: net.inet.ip.portrange.first and net.inet.ip.portrange.last.
IP_PORTRANGE_HIGH	use a high range of values, normally IPPORT_HIFIRSTAUTO and IPPORT_HILASTAUTO. This is adjustable through the sysctl setting: net.inet.ip.portrange.hifirst and net.inet.ip.portrange.hilast.
IP_PORTRANGE_LOW	use a low range of ports, which are normally restricted to privileged processes on UNIX systems. The range is normally from IPPORT_RESERVED down to 1 in descending order. This range is not sysctl-configurable.

Multicast Options

IP multicasting is supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on networks where the interface driver supports multicasting.

The IP_MULTICAST_TTL option changes the time-to-live (TTL) for outgoing multicast datagrams in order to control the scope of the multicasts:

```
u_char ttl;     /* range: 0 to 255, default = 1 */
setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Datagrams with a TTL of 1 are not forwarded beyond the local network. Multicast datagrams with a TTL of 0 will not be transmitted on any network, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than 1 may be forwarded to other networks if a multicast router is attached to the local network.

For hosts with multiple interfaces, each multicast transmission is sent from the primary network interface. The IP_MULTICAST_IF option overrides the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

where addr is the local IP address of the desired interface or INADDR_ANY to specify the default interface. An interface's local IP address and multicast capability can be obtained via the SIOCGIFCONF and SIOCGIFFLAGS ioctls. Normal applications should not need to use this option.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. The IP_MULTICAST_LOOP option gives the

sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;    /* 0 = disable, 1 = enable (default) */
setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

This option improves performance for applications that may have no more than one instance on a single host (such as a router daemon), by eliminating the overhead of receiving their own transmissions. It should generally not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

A host must become a member of a multicast group before it can receive datagrams sent to the group. To join a multicast group, use the IP_ADD_MEMBERSHIP option:

```
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

where mreq is the following structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* IP multicast address of group */
    struct in_addr imr_interface; /* local IP address of interface */
};
```

imr_interface should be INADDR_ANY to choose the default multicast interface, or the IP address of a particular multicast-capable interface if the host is multihomed. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface. Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where mreq contains the same values as used to add the membership. Memberships are dropped when the socket is closed or the process exits.

Raw IP Sockets

Raw IP sockets are connectionless, and are normally used with the sendto(2POSIX) and recvfrom(2POSIX) calls, though the connect(2POSIX) call may also be used to fix the destination for future packets (in which case the read(2POSIX) or recv(2POSIX) and write(2POSIX) or send(2POSIX) system calls may be used).

If proto is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If proto is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with), unless the IP_HDRINCL option has been set. Incoming packets are received with IP header and options intact.

IP_HDRINCL indicates the complete IP header is included with the data and may be used only with the SOCK_RAW type.

```
#include <netinet/ip.h>

int hincl = 1;                  /* 1 = on, 0 = off */
setsockopt(s, IPPROTO_IP, IP_HDRINCL, &hincl, sizeof(hincl));
```

Unlike previous releases, the program must set all the fields of the IP header, including the following:

```
ip->ip_v = IPVERSION;
ip->ip_hl = hlen >> 2;
ip->ip_id = 0;  /* 0 means kernel set appropriate value */
ip->ip_off = offset;
```

If the header source address is set to INADDR_ANY, the kernel will choose an appropriate address.

DIAGNOSTICS

A socket operation may fail and return one of the following error conditions:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket is not connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL]	An unknown socket option name was given.

[EINVAL] The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX), `send`(2POSIX), `recv`(2POSIX), `intro`(7P), `icmp`(7P), `inet`(7P)

NAME

route – kernel packet forwarding database

SYNOPSIS

```
#include <sys/socket.h>
#include <net/if.h>
#include <net/route.h>
int socket(PF_ROUTE, SOCK_RAW, int family);
```

DESCRIPTION

UNIX provides a number of packet routing facilities. The kernel maintains a routing information database, which is used in selecting the appropriate network interface when transmitting packets.

A user process (or multiple cooperating processes) maintains this database by sending messages over a special kind of socket. This replaces the fixed size *ioctl(2POSIX)* used in earlier releases. Routing table changes may only be carried out by the super user.

The operating system may spontaneously emit routing messages in response to external events, such as receipt of a redirect or failure to locate a suitable route for a request. The message types are described in greater detail below.

There are two types of routing database entries for a specific host, or for all hosts on a generic subnetwork (specified by a bit mask and value under the mask). A mask of all zeros defines a wildcard or default route. Hierarchical routes are supported.

When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. The protocol specifies the route through each interface as a direct connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests that the packet be sent to the same host as specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (in other words, the packet is forwarded).

When routing a packet, the kernel will attempt to find the most specific route to the destination. (If there are two different mask and value-under-the-mask pairs that match, the more specific is the one with more bits in the mask. A route to a host is assumed to be supplied with a mask of as many “1”’s as there are bits in the destination). If no entry is found, the destination is declared to be unreachable, and a routing-miss message is generated if there are any listeners on the routing control socket (described in more details below).

A wildcard routing entry is specified with a zero destination address value, and a mask of all zeroes. Wildcard routes will be used when the system fails to find other routes matching the destination. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

To open the channel for passing routing control messages, use the `socket()` call shown in the synopsis above.

The *family* parameter can be `AF_UNSPEC`, which will provide routing information for all address families. It can be restricted to a specific address family by specifying the one required. There can be more than one routing socket open per system.

Messages consist of a header followed by a number of sockadders (variable length, particularly in the `ISO` case) ; these are interpreted by position, and delimited by the new length entry in the sockaddr. An example of a message with four addresses might be an `ISO` redirect: Destination, Netmask, Gateway, and Author of the redirect. The interpretation of addresses present is provided by a bit mask within the header. The sequence runs from the least significant bit to the most significant bit within the vector.

Any messages sent to the kernel are returned, and copies are sent to all interested listeners. The kernel provides the process id for the sender, and the sender may use an additional sequence field to distinguish between outstanding messages. However, message replies may be lost when kernel buffers are exhausted.

The kernel may reject certain messages, and will indicate this by filling in the *rtm_errno* field. The routing code returns one of the following error conditions:

`EEXIST`	if requested to duplicate an existing entry,
`ESRCH`	if requested to delete a non-existent entry,
`ENOBUFS`	if insufficient resources were available to install a new route.

In the current implementation, all routing process run locally, and the values for *rtm_errno* are available through the normal *errno* mechanism, even if the routing reply message is lost.

A process may avoid reading replies to its own messages by issuing a *setsockopt(2POSIX)* call indicating that the *SO_USELOOPBACK* option at the *SOL_SOCKET* level is to be turned off. A process may ignore all messages from the routing socket by doing a *shutdown(2POSIX)* system call for further input.

If a route is in use when it is deleted, the routing entry will be marked down and removed from the routing table, but the resources associated with it will not be reclaimed until all references to it are released. User processes can obtain information about the routing entry to a specific destination by using a *RTM_GET* message, or by reading the */dev/kmem* device.

Messages include:

```
#define RTM_ADD         0x1    /* Add Route */
#define RTM_DELETE      0x2    /* Delete Route */
#define RTM_CHANGE      0x3    /* Change Metrics, Flags, or Gateway */
```

```
#define RTM_GET         0x4    /* Report Information */
#define RTM_LOOSING     0x5    /* Kernel Suspects Partitioning */
#define RTM_REDIRECT    0x6    /* Told to use different route */
#define RTM_MISS        0x7    /* Lookup failed on this address */
#define RTM_RESOLVE     0xb    /* Request to resolve dst to LL addr */
```

A message header consists of:

```
struct rt_msghdr {
       u_short  rtm_msglen;     /* to skip over non-understood messages*/
       u_char   rtm_version;    /* future binary compatibility */
       u_char   rtm_type;       /* message type */
       u_short  rmt_index;      /* index for associated ifp */
       int      rtm_flags;      /* flags, incl kern & message, e.g. DONE */
       int      rtm_addrs;      /* bit mask identifying sockaddrs in msg */
       pid_t    rmt_pid;        /* identify sender */
       int      rtm_seq;        /* for sender to identify action */
       int      rtm_errno;      /* why failed */
       int      rtm_use;        /* from rtentry */
       u_long   rtm_inits;      /* which values we are initializing */
       struct   rt_metrics rtm_rmx; /* metrics themselves */
};
```

where

```
struct rt_metrics {
        u_long  rmx_locks;      /* Kernel must leave these values alone*/
        u_long  rmx_mtu;        /* MTU for this path */
        u_long  rmx_hopcount;   /* max hops expected */
        u_long  rmx_expire;     /* lifetime for route, e.g. redirect */
        u_long  rmx_recvpipe;   /* inbound delay-bandwidth product */
        u_long  rmx_sendpipe;   /* outbound delay-bandwidth product */
        u_long  rmx_ssthresh;   /* outbound gateway buffer limit */
        u_long  rmx_rtt;        /* estimated round trip time */
        u_long  rmx_rttvar;     /* estimated rtt variance */
        u_long  rmx_pksent;     /* packets sent using this route */
        u_long  rmx_filler[4];  /* will be used for T/TCP later */
};
```

Values for `rtm_flags` include::

```
#define RTF_UP          0x1     /* route usable */
#define RTF_GATEWAY     0x2     /* destination is a gateway */
#define RTF_HOST        0x4     /* host entry (net otherwise) */
#define RTF_REJECT      0x8     /* host or net unreachable */
#define RTF_DYNAMIC     0x10    /* created dynamically (by redirect) */
#define RTF_MODIFIED    0x20    /* modified dynamically (by redirect) */
#define RTF_DONE        0x40    /* message confirmed */
#define RTF_MASK        0x80    /* subnet mask present */
#define RTF_CLONING     0x100   /* generate new routes on use */
#define RTF_XRESOLVE    0x200   /* external daemon resolves name */
#define RTF_LLINFO      0x400   /* generated by link layer (e.g. ARP) */
#define RTF_STATIC      0x800   /* manually added */
```

```
#define RTF_BLACKHOLE   0x1000     /* just discard pkts (during updates) */
#define RTF_PROTO2      0x4000     /* protocol specific routing flag */
#define RTF_PROTO1      0x8000     /* protocol specific routing flag */
#define RTF_PRCLONING   0x10000    /* protocol requires cloning */
#define RTF_WASCLONED   0x20000    /* route generated through cloning */
#define RTF_PROTO3      0x40000    /* protocol specific routing flag */
#define RTF_PINNED      0x100000   /* future use */
#define RTF_LOCAL       0x200000   /* route represents a local address */
#define RTF_BROADCAST   0x400000   /* route represents a bcast address */
#define RTF_MULTICAST   0x800000   /* route represents a mcast address */
```

Specifiers for metric values in `rmx_locks` and `rtm_inits` are:

```
#define RTV_MTU         0x1     /* init or lock _mtu */
#define RTV_HOPCOUNT    0x2     /* init or lock _hopcount */
#define RTV_EXPIRE      0x4     /* init or lock _hopcount */
#define RTV_RPIPE       0x8     /* init or lock _recvpipe */
#define RTV_SPIPE       0x10    /* init or lock _sendpipe */
#define RTV_SSTHRESH    0x20    /* init or lock _ssthresh */
#define RTV_RTT         0x40    /* init or lock _rtt */
#define RTV_RTTVAR      0x80    /* init or lock _rttvar */
```

Specifiers for which addresses are present in the messages are:

```
#define RTA_DST         0x1     /* destination sockaddr present */
#define RTA_GATEWAY     0x2     /* gateway sockaddr present */
#define RTA_NETMASK     0x4     /* netmask sockaddr present */
#define RTA_GENMASK     0x8     /* cloning mask sockaddr present */
#define RTA_IFP         0x10    /* interface name sockaddr present */
#define RTA_IFA         0x20    /* interface addr sockaddr present */
#define RTA_AUTHOR      0x40    /* sockaddr for author of redirect */
#define RTA_BRD         0x80    /* for NEWADDR, broadcast or p-p dest addr */
```

SEE ALSO

`route`(1M)

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tcp – Internet Transmission Control Protocol

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int socket(AF_INET, SOCK_STREAM, 0);
```

DESCRIPTION

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses". Each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets using the TCP protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default, TCP sockets are created active. To create a passive socket the `listen`(2POSIX) system call must be used after binding the socket using the *bind*(2POSIX) system call. Only passive sockets may use the `accept`(2POSIX) call to accept incoming connections. Only active sockets may use the *connect*(2POSIX) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, termed "wildcard addressing", allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received. This address usually corresponds to the peer entity's network.

TCP supports a number of socket options which can be set with `setsockopt`(2POSIX) and tested with `getsockopt`(2POSIX):

TCP_NODELAY — Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as window systems that send a stream of mouse events which receive no replies, this packetization may cause significant delays. The boolean option `TCP_NODELAY` defeats this algorithm.

TCP_MAXSEG	By default, a sender- and receiver-TCP will negotiate among themselves to determine the maximum segment size to be used for each connection. The `TCP_MAXSEG` option allows the user to determine the result of this negotiation, and to reduce it if desired.
TCP_NOOPT	TCP usually sends a number of options in each packet, cor responding to various TCP extensions which are provided in this implementation. The boolean option `TCP_NOOPT` is provided to disable TCP option use on a per-connection basis.
TCP_NOPUSH	By convention, the sender-TCP will set the "push" bit and begin transmission immediately (if permitted) at the end of every user call to `write`(2POSIX) or `writev`(2POSIX). When the option is set to a non-zero value, TCP will delay sending any data at all until either the socket is closed, or the internal send buffer is filled.

The option level for the `setsockopt`(2POSIX) call is the protocol number for TCP, available from `getprotobyname`(3POSIX), or `IPPROTO_TCP`. All options are declared in `<netinet/tcp.h>`.

Options at the IP transport level may be used with TCP; see `ip`(7P). Incoming connection requests that are source-routed are noted, and the reverse source route is used when responding.

MIB VARIABLES

The TCP protocol implements three variables in the `net.inet` branch of the `sysctl`(3POSIX) MIB.

TCPCTL_DO_RFC1323	(`tcp.rfc1323`) Implement the window scaling and timestamp options of RFC 1323 (default `true`).
TCPCTL_DO_RFC1644	(`tcp.rfc1644`) Implement Transaction TCP, as in RFC 1644.
TCPCTL_MSSDFLT	(`tcp.mssdflt`) The default value used for the maximum segment size (MSS) when no advice to the contrary is received from MSS negotiation (default `512`).

DIAGNOSTICS

A socket operation may fail and return one of the following error conditions:

[EISCONN]	when trying to establish a connection on a socket which already has one;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[ETIMEDOUT]	when a connection was dropped due to excessive retransmissions;
[ECONNRESET]	when the remote peer forces the connection to be closed;
[ECONNREFUSED]	when the remote peer actively refuses to establish a connection (usually because no process is listening to the port);
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.
[EAFNOSUPPORT]	when an attempt is made to create a bind or connect a socket to a multicast address.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX), `socket`(2POSIX), `sysctl`(3POSIX), `inet`(7P), `intro`(7P), `ip`(7P)

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

int **socket**(*AF_INET*, *SOCK_DGRAM*, *0*);

DESCRIPTION

The UDP datagram protocol is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the `sendto`(2POSIX) and `recvfrom`(2POSIX) calls. The `connect`(2POSIX) call may also be used to fix the destination for future packets (in which case the `recv`(2POSIX) or `read`(2POSIX) and `send`(2POSIX) or `write`(2POSIX) system calls may be used).

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (a UDP port cannot be “connected” to a TCP port). In addition, broadcast packets may be sent (assuming the underlying network supports this) by using a reserved “broadcast address”; this address is network—interface—dependent.

Options at the IP transport level may be used with UDP; see `ip`(7P).

DIAGNOSTICS

A socket operation may fail and return one of the following error conditions:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn’t been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsockopt`(2POSIX), `recv`(2POSIX), `send`(2POSIX), `socket`(2POSIX), `intro`(7P), `inet`(7P), `ip`(7P)

Index

A

arp — Address Resolution Protocol 13

I

icmp — Internet Control Message Protocol 15
inet — Internet protocol family 16
intro — introduction to networking facilities 11
ip — Internet Protocol 20

R

route — kernel packet forwarding database 25

T

tcp — Internet Transmission Control Protocol 29

U

udp — Internet User Datagram Protocol 32