AT&T UNIX® PC
ENHANCED TCP/IP
WIN/3B LAN INTERFACE

PROGRAMMER's
REFERENCE MANUAL

July 1986

RESTRICTED RIGHTS

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b) (3) (B) of the Rights in Technical Data and Computer Software clause in DAR 7-104.9(a). The Wollongong Group, Inc., 1129 San Antonio Road, Palo Alto, California 94303.

TRADEMARK ACKNOWLEDGEMENT

UNIX is a registered trademark of AT&T.

WIN is a trademark of The Wollongong Group, Inc.

DEC is a trademark of Digital Equipment Corporation.

TABLE OF CONTENTS

3T. Transport Level Interface Library

3W. Socket Compatability Library

4. Protocols and Drivers

5. Data Files

NAME

intro – introduction to general user commands.

DESCRIPTION

This section contains information related to the general user commands. General user commands and their functions are:

Command	*Description*
finger	user information lookup program
ftp	file transfer program
hostid	set or print identifier of current host system
hostname	set or print name of current host system
mailx	interactive message processing system
netstat	show network status
rcp	remote file copy
remsh	remote shell
rlogin	remote login
ruptime	show host status of local machines
rwho	who is logged in on local machines
telnet	user interface to the TELNET protocol
tftp	trivial file transfer program

NAME

finger – user information lookup program

SYNOPSIS

finger [options] name ...
finger [user] @host

DESCRIPTION

By default *finger* lists the login name, full name, terminal name and write status (as a "*" before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current UNIX user. (Idle time is minutes if it is a single integer, hours and minutes if a ":" is present, or days and hours if a "d" is present.)

A longer format also exists and is used by *finger* whenever a list of peoples names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file *.plan* in their home directory, and the project on which they are working from the file *.project* also in the home directory.

Finger options include:

- **b** Briefer long form list of users.
- **f** Suppress heading in the short and quick output format.
- **h** Suppress printing of the *.project* files.
- **i** Same as quick list but includes idle time.
- **l** Force long output format.
- **m** Match arguments only on user name.

- **-p** Suppress printing of the *.plan* files
- **-q** Quick list with only login name, terminal name and login time.
- **-s** Force short output format.
- **-w** Suppress printing of the full name in the short output format.

Finger can also be used as a network protocol by specifying a hostname or address or a specific user at a host. *Finger* will display information about a user or users on that host. Instead of the username, any of the above options can be specified at a host.

FILES

/etc/utmp	who file
/etc/passwd	for users names, offices, ...
~/.plan	plans
~/.project	projects

NAME

ftp – file transfer program

SYNOPSIS

ftp [– v] [– d] [– i] [– n] [– g] [host]

DESCRIPTION

Ftp is the user interface to the ARPANET standard File Transfer Protocol (FTP). The program allows a user to transfer files to and from a remote network site.

The client host with which *ftp* is to communicate may be specified on the command line. If this is done, *ftp* will immediately attempt to establish a connection to an FTP server on that host; otherwise, *ftp* will enter its command interpreter and await instructions from the user. When *ftp* is awaiting commands from the user the prompt "ftp>" is provided the user. If insufficient command arguments are supplied by the user, *ftp* will prompt for them.

Ftp may be interrupted, typically by striking the "del" key or ctrl-C. If this is done while *ftp* is attempting to carry out a command, *ftp* will revert to the command interpreter and display the "ftp>" prompt. Otherwise, *ftp* will be terminated.

The following commands are recognized by *ftp*. They may be shortened so long as they remain unique.

! Invoke a shell on the local machine.

append *local-file* [*remote-file*]

Append *local-file* to a file on the remote machine. If *remote-file* is left unspecified, the name of *local-file* is used in naming *remote-file*. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII. This is the default *type*.

bell Arrange that a bell be sounded after each file transfer command is completed.

binary Set the file transfer *type* to support *binary* image transfer.

bye Terminate the FTP session with the remote server and exit *ftp*.

cd *remote-directory*
Change the working directory on the remote machine to *remote-directory*.

close Terminate the FTP session with the remote server, and return to the command interpreter.

delete *remote-file*
Delete the file *remote-file* on the remote machine.

debug Toggle debugging mode. When debugging is on, *ftp* prints each command sent to the remote machine, preceded by the string "-->". For a list of the commands, see *ftpd(8)*.

dir [*remote-directory*] [*local-file*]
Print a listing of the contents of *remote-directory* and, optionally, place the output in *local-file*. If no directory is specified, the current working directory on the remote machine is used. If *local-file* is not specified, output comes to the terminal.

form *format*
Set the file transfer *form* to *format*. The default format is "non-print". At this printing, only the default form is supported.

get *remote-file* [*local-file*]
Retrieve *remote-file* and store it on the local machine. If the name of *local-file* is not specified, it is given the same name it has on the remote machine. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

hash Toggle hash-sign ("#") printing for each data block transferred. The size of a data block is 4096 bytes.

glob Toggle local file name globbing. With file name globbing enabled, each local file or pathname is processed for *sh(1)* metacharacters "*?[]~". An additional pair of metacharacters, "{}", is also processed. This pair may enclose several comma-separated strings for each of which a match is sought. With globbing disabled all local files and pathnames are treated literally. Globbing is always on with reference to remote files.

help [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *ftp* prints a list of the known commands.

lcd [*directory*]
Change the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
Print an abbreviated listing of the contents of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If *local-file* is not specified,

the output is sent to the terminal.

mdelete *remote-files*
Delete the specified files on the remote machine. If globbing is enabled, the specification of *remote-files* will first be expanded using *ls*.

mdir *remote-files local-file*
Obtain a directory listing of multiple files on the remote machine and place the result in *local-file*.

mget *remote-files*
Retrieve the specified files from the remote machine and place them in the current local directory. If globbing is enabled, the specification of remote files will first be expanded using *ls*.

mkdir *directory-name*
Make a directory on the remote machine.

mls *remote-files local-file*
Obtain an abbreviated listing of multiple files on the remote machine and place the result in *local-file*.

mode [*mode-name*]
Set the file transfer *mode* to *mode-name*. The default mode is "stream" mode. At this printing, only the default is supported.

mput *local-files*
Transfer multiple *local-files* from the current local directory to the current working directory on the remote machine.

open *host* [*port*]

Establish a connection to the specified *host* FTP server. An optional *port* number may be supplied, in which case, *ftp* will attempt to contact an FTP server at that port. If autologin is enabled (default), *ftp* will also attempt to automatically log the user in to the FTP server (see below).

prompt Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default), any *mget* or *mput* will transfer all files.

put *local-file* [*remote-file*]
Store *local-file* on the remote machine. If *remote-file* is left unspecified, the name of *local-file* is used in naming *remote-file*. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for bye.

quote *arg1 arg2* ...
The arguments specified are sent, verbatim, to the remote FTP server. A single FTP reply code is expected in return.

recv *remote-file* [*local-file*]
A synonym for get.

remotehelp [*command-name*]
Request help from the remote FTP server. If a *command-name* is specified it is supplied to the server as well.

rename [*from*] [*to*]
Rename the file *from* on the remote machine, to the file *to*.

rmdir *directory-name*
Delete a directory on the remote machine.

send *local-file* [*remote-file*]
A synonym for put.

sendport
Toggle the use of PORT commands. By default, *ftp* will attempt to use a PORT command when establishing a connection for each data transfer. If the PORT command fails, *ftp* will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

status Show the current status of *ftp*.

struct [*struct-name*]
Set the file transfer *structure* to *struct-name*. By default "file" structure is used. At this printing, only the default is supported.

tenex Set the file transfer *type* to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer *type* to *type-name*. If *type* is not specified, the current type is printed. The three valid types are "tenex", "binary" and "ascii", which is the default.

user *user-name* [*password*] [*account*]

Identify yourself to the remote FTP server. If *password* is not specified and the server requires it, *ftp* will prompt the user for it (after disabling local echo). If *account* is not specified, and the FTP server requires it, the user will be prompted for it. Unless *ftp* is invoked with autologin disabled, this process is done automatically on initial connection to the FTP server.

verbose

Toggle *verbose* mode. In *verbose* mode, all responses from the FTP server are displayed to the user. In addition, if *verbose* is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, *verbose* is off.

? [*command*]

A synonym for help.

Command arguments which have embedded spaces may be quoted with quote (") marks.

FILE NAMING CONVENTIONS

Files specified as arguments to *ftp* commands are processed according to the following rules.

1) If the file name "- " is specified, the **stdin** (for reading) or **stdout** (for writing) is used.

2) If the first character of the file name is "|", the remainder of the argument is interpreted as a shell command. *Ftp* then forks a shell with the argument supplied, and reads (writes) from the *stdout* (stdin). If the shell command includes spaces, the argument must be quoted; e.g. ""|ls

-lt"'. A particularly useful example of this mechanism is: "dir < directory name> þg".

3) Failing the above checks, if "globbing" is enabled, local file names are expanded as per the *glob* command.

OPTIONS

Options may be specified at the command line or to the command interpreter.

The – v (verbose on) option forces *ftp* to show all responses from the remote server, as well as report on data transfer statistics.

The – n option restrains *ftp* from attempting autologin upon initial connection. If auto-login is enabled, *ftp* will check the *.netrc(5)* file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* will use the login name on the local machine as the user identity on the remote machine, and prompt for a password and, optionally, an account with which to login.

The – i option turns off interactive prompting during multiple file transfers.

The – d option enables debugging.

The – g option disables file name globbing.

BUGS

Many FTP server implementations do not support operations such as "print working directory". "mget" and "mdelete" commands should be used with caution. Specifying a directory where a plain file name is expected could produce unexpected results.

NAME

hostid – set or print identifier of current host system

SYNOPSIS

hostid [identifier]

DESCRIPTION

The *hostid* command sets or prints the *identifier* of the current host in hexadecimal. By default, the value of *identifier* is 0. This numeric value is expected to be unique across all hosts and is normally set to the host's Internet address. The superuser can set the *identifier* by giving a hexadecimal argument; this is usually done in the startup script, */etc/iddrv/ether.rc*

SEE ALSO

gethostid(3W), sethostid(3W)

NAME

hostname – set or print name of current host system

SYNOPSIS

hostname [nameofhost]

DESCRIPTION

The *hostname* command prints the name of the current host. A user with superuser privileges can set the *hostname* by giving an argument.

SEE ALSO

gethostname(3W), sethostname(3W)

NAME

mailx – interactive message processing system

SYNOPSIS

mailx [*options*] [*name...*]

DESCRIPTION

The command *mailx* provides a comfortable, flexible environment for sending and receiving messages electronically. When reading mail, *mailx* provides commands to facilitate saving, deleting, and responding to messages. When sending mail, *mailx* allows editing, reviewing and other modification of the message as it is entered.

Incoming mail is stored in a standard file for each user, called the system *mailbox* for that user. When *mailx* is called to read messages, the *mailbox* is the default place to find them. As messages are read, they are marked to be moved to a secondary file for storage, unless specific action is taken, so that the messages need not be seen again. This secondary file is called the *mbox* and is normally located in the user's HOME directory (see "MBOX" (ENVIRONMENT VARIABLES) for a description of this file). Messages remain in this file until forcibly removed.

On the command line, *options* start with a dash (–) and any other arguments are taken to be destinations (recipients). If no recipients are specified, *mailx* will attempt to read messages from the *mailbox*. Command line options are:

– d	Turn on debugging output. Neither particularly interesting nor recommended.
– e	Test for presence of mail.

Mailx prints nothing and exits with a successful return code if there is mail to read.

-f [*filename*] Read messages from *filename* instead of *mailbox*. If no *filename* is specified, the *mbox* is used.

-F Record the message in a file named after the first recipient. Overrides the "record" variable, if set (see ENVIRONMENT VARIABLES).

-h *number* The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.

-H Print header summary only.

-i Ignore interrupts. See also "ignore" (ENVIRONMENT VARIABLES).

-n Do not initialize from the system default *Mailx.rc* file.

-N Do not print initial header summary.

-r *address* Pass *address* to network delivery software. All tilde commands are disabled.

-s *subject* Set the Subject header field to *subject*.

-u *user* Read *user*'s *mailbox*. This is only effective if *user*'s *mailbox* is not read protected.

-U Convert *uucp* style addresses to internet standards. Overrides

the "conv" environment variable.

When reading mail, *mailx* is in *command mode*. A header summary of the first several messages is displayed, followed by a prompt indicating *mailx* can accept regular commands (see COMMANDS below). When sending mail, *mailx* is in *input mode*. If no subject is specified on the command line, a prompt for the subject is printed. As the message is typed, *mailx* will read the message and store it in a temporary file. Commands may be entered by beginning a line with the tilde (~) escape character followed by a single command letter and optional arguments. See TILDE ESCAPES for a summary of these commands.

At any time, the behavior of *mailx* is governed by a set of *environment variables*. These are flags and valued parameters which are set and cleared via the **set** and **unset** commands. See ENVIRONMENT VARIABLES below for a summary of these parameters.

Recipients listed on the command line may be of three types: login names, shell commands, or alias groups. Login names may be any network address, including mixed network addressing. If the recipient name begins with a pipe symbol (|), the rest of the name is taken to be a shell command to pipe the message through. This provides an automatic interface with any program that reads the standard input, such as *lp*(1) for recording outgoing mail on paper. Alias groups are set by the alias command (see COMMANDS below) and are lists of recipients of any type.

Regular commands are of the form

[**command**] [*msglist*] [*arguments*]

If no command is specified in *command mode*, print is assumed. In *input mode*, commands are recognized by the escape character, and lines not treated as commands are taken as input for the message.

Each message is assigned a sequential number, and there is at any time the notion of a 'current' message, marked by a '>' in the header summary. Many commands take an optional list of messages (*msglist*) to operate on, which defaults to the current message. A *msglist* is a list of message specifications separated by spaces, which may include:

n	Message number **n**.
.	The current message.
^	The first undeleted message.
$	The last message.
*****	All messages.
n– m	An inclusive range of message numbers.
user	All messages from **user**.
/string	All messages with **string** in the subject line (case ignored).
:c	All messages of type *c*, where *c* is one of:

d	deleted messages
n	new messages
o	old messages
r	read messages
u	unread messages

Note that the context of the command determines whether this type of

message specification makes sense.

Other arguments are usually arbitrary strings whose usage depends on the command involved. File names, where expected, are expanded via the normal shell conventions (see *sh*(1)). Special characters are recognized by certain commands and are documented with the commands below.

At start-up time, *mailx* reads commands from a system-wide file (**/usr/lib/mailx/mailx.rc**) to initialize certain parameters, then from a private start-up file (**$HOME/.mailrc**) for personalized variables. Most regular commands are legal inside start-up files, the most common use being to set up initial display options and alias lists. The following commands are not legal in the start-up file: **!**, **Copy**, **edit**, **followup**, **Followup**, **hold**, **mail**, **preserve**, **reply**, **Reply**, **shell**, and **visual**. Any errors in the start-up file cause the remaining lines in the file to be ignored.

COMMANDS

The following is a complete list of *mailx* commands:

!*shell-command*

Escape to the shell. See "SHELL" (ENVIRONMENT VARIABLES).

*comment*

Null command (comment). This may be useful in *.mailrc* files.

=

Print the current message number.

?

Prints a summary of commands.

alias *alias name ...*
group *alias name ...*

Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

alternates *name ...*

Declares a list of alternate names for your login. When responding to a message, these names are removed from the list of recipients for the response. With no arguments, alternates prints the current list of alternate names. See also "allnet" (ENVIRONMENT VARIABLES).

cd [*directory*]
chdir [*directory*]

Change directory. If *directory* is not specified, $HOME is used.

copy [*filename*]
copy [*msglist*] *filename*

Copy messages to the file without marking the messages as saved. Otherwise equivalent to the save command.

Copy [*msglist*]

Save the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the Save

command.

delete [*msglist*]
Delete messages from the *mailbox*. If "autoprint" is set, the next message after the last one deleted is printed (see ENVIRONMENT VARIABLES).

discard [*header-field* ...]
ignore [*header-field* ...]
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." The fields are included when the message is saved. The Print and Type commands override this command.

dp [*msglist*]
dt [*msglist*]
Delete the specified messages from the *mailbox* and print the next message after the last one deleted. Roughly equivalent to a delete command followed by a print command.

echo *string* ...
Echo the given strings (like *echo*(1)).

edit [*msglist*]
Edit the given messages. The messages are placed in a temporary file and the "EDITOR" variable is used to get the name of the editor (see ENVIRONMENT VARIABLES). Default editor is *ed*(1).

exit
xit

Exit from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also quit).

file [*filename*]
folder [*filename*]

Quit from the current file of messages and read in the specified file. Several special characters are recognized when used as file names, with the following substitutions:

%	the current *mailbox*.
%user	the *mailbox* for user.
#	the previous file.
&	the current *mbox*.

Default file is the current *mailbox*.

folders

Print the names of the files in the directory set by the "folder" variable (see ENVIRONMENT VARIABLES).

followup [*message*]

Respond to a message, recording the response in a file whose name is derived from the author of the message. Overrides the "record" variable, if set. See also the Followup, Save, and Copy commands and "outfolder" (ENVIRONMENT VARIABLES).

Followup [*msglist*]
Respond to the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line is taken from the first message and the response is recorded in a file whose name is derived from the author of the first message. See also the **followup**, **Save**, and **Copy** commands and "outfolder" (ENVIRONMENT VARIABLES).

from [*msglist*]
Prints the header summary for the specified messages.

group *alias name ...*
alias *alias name ...*
Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

headers [*message*]

Prints the page of headers which includes the message specified. The "screen" variable sets the number of headers per page (see ENVIRONMENT VARIABLES). See also the **z** command.

help
Prints a summary of commands.

hold [*msglist*]
preserve [*msglist*]
Holds the specified messages in the *mailbox*.

if *s* | *r*
mail-commands
else
mail-commands
endif

Conditional execution, where *s* will execute following *mail-commands*, up to an else or endif, if the program is in *send* mode, and *r* causes the *mail-commands* to be executed only in *receive* mode. Useful in the *.mailrc* file.

ignore *header-field* ...
discard *header-field* ...

Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." All fields are included when the message is saved. The Print and Type commands override this command.

list

Prints all commands available. No explanation is given.

mail *name* ...

Mail a message to the specified users.

mbox [*msglist*]

Arrange for the given messages to end up in the standard *mbox* save file when *mailx* terminates normally. See "MBOX" (ENVIRONMENT VARIABLES) for a description of this file. See also the exit and quit commands.

next [*message*]

Go to next message matching *message*. A *msglist* may be specified, but in this case the first valid message in the list is the only one used. This is useful for jumping to the next message from a specific user, since the name would be taken as a command in the absence of a real command. See the discussion of *msglists* above for a description of possible message specifications.

pipe [*msglist*] [*shell-command*]

|[*msglist*] [*shell-command*]

Pipe the message through the given *shell-command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the "cmd" variable. If the "page" variable is set, a form feed character is inserted after each message (see ENVIRONMENT VARIABLES).

preserve [*msglist*]

hold [*msglist*]

Preserve the specified messages in the *mailbox*.

Print [*msglist*]
Type [*msglist*]

Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the Ignore command.

print [*msglist*]

type [*msglist*]
Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command specified by the "PAGER" variable. The default command is *pg*(1) (see ENVIRONMENT VARIABLES).

quit
Exit from *mailx*, storing messages that were read in *mbox* and unread messages in the *mailbox*. Messages that have been explicitly saved in a file are deleted.

Reply [*msglist*]

Respond [*msglist*]
Send a response to the author of each message in the *msglist*. The subject line is taken from the first message. If "record" is set to a filename, the response is saved at the end of that file (see ENVIRONMENT VARIABLES).

reply [*message*]

respond [*message*]
Reply to the specified message, including all other recipients of the message. If "record" is set to a filename, the response is saved at the end of that file (see ENVIRONMENT VARIABLES).

Save [*msglist*]

Save the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the **Copy**, **followup**, and **Followup** commands and "outfolder" (ENVIRONMENT VARIABLES).

save [*filename*]

save [*msglist*] *filename*

Save the specified messages in the given file. The file is created if it does not exist. The message is deleted from the *mailbox* when *mailx* terminates unless "keepsave" is set (see also ENVIRONMENT VARIABLES and the **exit** and **quit** commands).

set
set *name*
set *name*= *string*
set *name*= *number*

Define a variable called *name*. The variable may be given a null, string, or numeric value. **Set** by itself prints all defined variables and their values. See ENVIRONMENT VARIABLES for detailed descriptions of the *mailx* variables.

shell

Invoke an interactive shell (see also "SHELL" (ENVIRONMENT VARIABLES)).

size [*msglist*]
Print the size in characters of the specified messages.

source *filename*
Read commands from the given file and return to command mode.

top [*msglist*]
Print the top few lines of the specified messages. If the "toplines" variable is set, it is taken as the number of lines to print (see ENVIRONMENT VARIABLES). The default is 5.

touch [*msglist*]
Touch the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the *mbox* upon normal termination. See exit and quit.

Type [*msglist*]

Print [*msglist*]
Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the ignore command.

type [*msglist*]

print [*msglist*]
Print the specified messages. If "crt" is set, the messages longer than the number of lines

specified by the "crt" variable are paged through the command specified by the "PAGER" variable. The default command is *pg*(1) (see ENVIRONMENT VARIABLES).

undelete [*msglist*]
Restore the specified deleted messages. Will only restore messages deleted in the current mail session. If "autoprint" is set, the last message of those restored is printed (see ENVIRONMENT VARIABLES).

unset *name ...*
Causes the specified variables to be erased. If the variable was imported from the execution environment (i.e., a shell variable) then it cannot be erased.

version
Prints the current version and release date.

visual [*msglist*]
Edit the given messages with a screen editor. The messages are placed in a temporary file and the "VISUAL" variable is used to get the name of the editor (see ENVIRONMENT VARIABLES).

write [*msglist*] *filename*
Write the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the save command.

xit
exit

Exit from *mailx*, without changing the *mailbox*. No messages are saved in the *mbox* (see also quit).

z[+ |-]

Scroll the header display forward or backward one screen– full. The number of headers displayed is set by the "screen" variable (see ENVIRONMENT VARIABLES).

TILDE ESCAPES

The following commands may be entered only from *input mode*, by beginning a line with the tilde escape character (~). See "escape" (ENVIRONMENT VARIABLES) for changing this special character.

~! *shell-command*

Escape to the shell.

~.

Simulate end of file (terminate message input).

~: *mail-command*
~_ *mail-command*

Perform the command-level request. Valid only when sending a message while reading mail.

~?

Print a summary of tilde escapes.

~A

Insert the autograph string "Sign" into the message (see ENVIRONMENT VARIABLES).

~a

Insert the autograph string "sign" into the message (see ENVIRONMENT VARIABLES).

~b *name ...*

Add the *names* to the blind carbon copy (Bcc) list.

~c *name ...*

Add the *names* to the carbon copy (Cc) list.

~d

Read in the *dead.letter* file. See "DEAD" (ENVIRONMENT VARIABLES) for a description of this file.

~e

Invoke the editor on the partial message. See also "EDITOR" (ENVIRONMENT VARIABLES).

~f [*msglist*]

Forward the specified messages. The messages are inserted into the message, without alteration.

~h

Prompt for Subject line and To, Cc, and Bcc lists. If the field is displayed with an initial value, it may be edited as if you had just typed it.

~i *string*

Insert the value of the named variable into the text of the message. For example, ~A is

equivalent to '~I Sign.'

~m [*msglist*]
Insert the specified messages into the letter, shifting the new text to the right one tab stop. Valid only when sending a message while reading mail.

~p
Print the message being entered.

~q
Quit from input mode by simulating an interrupt. If the body of the message is not null, the partial message is saved in *dead.letter*. See "DEAD" (ENVIRONMENT VARIABLES) for a description of this file.

~r *filename*
~< *filename*
~< !*shell-command*
Read in the specified file. If the argument begins with an exclamation point (!), the rest of the string is taken as an arbitrary shell command and is executed, with the standard output inserted into the message.

~s *string* ...
Set the subject line to *string*.

~t *name* ...
Add the given *names* to the To list.

~v

Invoke a preferred screen editor on the partial message. See also "VISUAL" (ENVIRONMENT VARIABLES).

~w *filename*

Write the partial message onto the given file, without the header.

~x

Exit as with ~q except the message is not saved in *dead.letter*.

~|*shell-command*

Pipe the body of the message through the given *shell-command*. If the *shell-command* returns a successful exit status, the output of the command replaces the message.

ENVIRONMENT VARIABLES

The following are environment variables taken from the execution environment and are not alterable within *mailx*.

HOME= *directory*

The user's base of operations.

MAILRC= *filename*

The name of the start-up file. Default is $HOME/.mailrc.

The following variables are internal *mailx* variables. They may be imported from the execution environment or set via the **set** command at any time. The **unset** command may be used to erase variables.

allnet

All network names whose last component (login name) match are treated as identical. This causes the *msglist* message specifications to behave similarly. Default is **noallnet**. See also the **alternates** command and the "metoo" variable.

append

Upon termination, append messages to the end of the *mbox* file instead of prepending them. Default is **noappend**.

askcc

Prompt for the Cc list after message is entered. Default is **noaskcc**.

asksub

Prompt for subject if it is not specified on the command line with the **-s** option. Enabled by default.

autoprint

Enable automatic printing of messages after **delete** and **undelete** commands. Default is **noautoprint**.

bang

Enable the special-casing of exclamation points (!) in shell escape command lines as in *vi*(1). Default is **nobang**.

cmd= *shell-command*

Set the default command for the **pipe**

command. No default value.

conv=*conversion*
Convert uucp addresses to the specified address style. The only valid conversion now is *internet*, which requires a mail delivery program conforming to the RFC822 standard for electronic mail addressing. Conversion is disabled by default. See also "sendmail" and the **-U** command line option.

crt=*number*
Pipe messages having more than *number* lines through the command specified by the value of the "PAGER" variable (*pg*(1) by default). Disabled by default.

DEAD=*filename*
The name of the file in which to save partial letters in case of untimely interrupt or delivery errors. Default is $HOME/dead.letter.

debug
Enable verbose diagnostics for debugging. Messages are not delivered. Default is **nodebug**.

dot
Take a period on a line by itself during input from a terminal as end-of-file. Default is **nodot**.

EDITOR=*shell-command*
The command to run when the edit or ~e

command is used. Default is *ed*(1).

escape= *c*
Substitute *c* for the ~ escape character.

folder= *directory*
The directory for saving standard mail files. User specified file names beginning with a plus (+) are expanded by preceding the filename with this directory name to obtain the real filename. If *directory* does not start with a slash (/), $HOME is prepended to it. In order to use the plus (+) construct on a *mailx* command line, "folder" must be an exported *sh* environment variable. There is no default for the "folder" variable. See also "outfolder" below.

header
Enable printing of the header summary when entering *mailx*. Enabled by default.

hold
Preserve all messages that are read in the *mailbox* instead of putting them in the standard *mbox* save file. Default is **nohold**.

ignore
Ignore interrupts while entering messages. Handy for noisy dial-up lines. Default is **noignore**.

ignoreeof
Ignore end-of-file during message input. Input must be terminated by a period (.) on a line by

itself

or by the ". command. Default is **noignoreeof**. See also "dot" above.

keep

When the *mailbox* is empty, truncate it to zero length instead of removing it. Disabled by default.

keepsave

Keep messages that have been saved in other files in the *mailbox* instead of deleting them. Default is **nokeepsave**.

MBOX=*filename*

The name of the file to save messages which have been read. The **xit** command overrides this function, as does saving the message explicitly in another file. Default is $HOME/mbox.

metoo

If your login appears as a recipient, do not delete it from the list. Default is **nometoo**.

LISTER= *shell-command*

The command (and options) to use when listing the contents of the "folder" directory. The default is *ls*(1).

onehop

When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be

relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away).

outfolder

Causes the files used to record outgoing messages to be located in the directory specified by the "folder" variable unless the pathname is absolute. Default is **nooutfolder**. See "folder" above and the Save, Copy, followup, and Followup commands.

page

Used with the pipe command to insert a form feed after each message sent through the pipe. Default is **nopage**.

PAGER= *shell-command*

The command to use as a filter for paginating output. This can also be used to specify the options to be used. Default is *pg*(1).

prompt= *string*

Set the *command mode* prompt to *string*. Default is "? ".

quiet

Refrain from printing the opening message and version when entering *mailx*. Default is **noquiet**.

record=*filename*
Record all outgoing mail in *filename*. Disabled by default. See also "outfolder" above.

save
Enable saving of messages in *dead.letter* on interrupt or delivery error. See "DEAD" for a description of this file. Enabled by default.

screen=*number*
Sets the number of lines in a screen- full of headers for the headers command.

sendmail=*shell-command*
Alternate command for delivering messages. Default is *mail*(1).

sendwait
Wait for background mailer to finish before returning. Default is **nosendwait**.

SHELL=*shell-command*
The name of a preferred command interpreter. Default is *sh*(1).

showto
When displaying the header summary and the message is from you, print the recipient's name instead of the author's name.

sign=*string*
The variable inserted into the text of a message when the ~a (autograph) command is given. No default (see also ~I (TILDE ESCAPES)).

Sign = *string*
The variable inserted into the text of a message when the ~A command is given. No default (see also ~I (TILDE ESCAPES)).

toplines = *number*
The number of lines of header to print with the **top** command. Default is 5.

VISUAL = *shell-command*
The name of a preferred screen editor. Default is *vi*(1).

FILES

$HOME/.mailrc	personal start-up file
$HOME/mbox	secondary storage file
/usr/mail/*	post office directory
/usr/lib/mailx/mailx.help*	help message files
/usr/lib/mailx/mailx.rc	global start-up file
/tmp/R[emqsx]*	temporary files

SEE ALSO

mail(1), *pg(1)*, *ls(1)*, UNIX System V User Manual

BUGS

Where *shell-command* is shown as valid, arguments are not always allowed. Experimentation is recommended.

Internal variables imported from the execution environment cannot be **unset**.

The full internet addressing is not fully supported by *mailx*. The new standards need some time to settle down.

Attempts to send a message having a line consisting only of a "." are treated as the end of the message by *mail*(1) (the standard mail delivery program).

NAME

netstat – show network status

SYNOPSIS

netstat [– **Aahlmnrst**] [*interval*] [*system* [*core*]]

DESCRIPTION

The *netstat* command symbolically displays the contents of various network-related data structures. The options have the following meaning:

– **A** show the address of any associated protocol control blocks; used for debugging

– **a** show the state of all sockets; normally sockets used by server processes are not shown

– **h** show the state of the IMP host table

– **i** show the state of interfaces which have been auto-configured (interfaces statically configured into a system, but not located at boot time are not shown)

– **m** show statistics recorded by the memory management routines (the network manages a "private share" of memory)

– **n** show network addresses as numbers (normally *netstat* interprets addresses and attempts to display them symbolically)

– **r** show the routing tables

– **s** show per-protocol statistics

– **t** show the timer value (when used with – *i* option)

The arguments, *system* and *core* allow substitutes for the defaults "/unix" and "/dev/kmem".

If an *interval* is specified, *netstat* will continuously display the information regarding packet traffic on the configured network interfaces, pausing *interval* seconds before refreshing the screen.

There are a number of display formats, depending on the information presented. The default display, for active sockets, shows the local and remote addresses, send and receive queue sizes (in bytes), protocol, and, optionally, the internal state of the protocol.

Address formats are of the form "host.port" or "network.port" if a socket's address specifies a network but no specific host address. When known the host and network addresses are displayed symbolically according to the data bases *hosts(5)* and *networks(5)*, respectively. If a symbolic name for an address is unknown, or if the **-n** option is specified, the address is printed in the Internet "dot format"; refer to *inet*(3N) for more information regarding this format. Unspecified, or "wildcard", addresses and ports appear as "*".

The interface display provides a table of cumulative statistics regarding packets transferred, errors, and collisions. The network address (currently Internet specific) of the interface and the maximum transmission unit in bytes ("mtu") are also displayed.

The routing table display indicates the available routes and their status. Each route consists of a destination host or network and a gateway to use in forwarding packets. The flags field shows the state of the route ("U" if "up"), and whether the route is to a gateway ("G"). Direct routes are created for each interface attached to the local host. The refcnt field gives the current number of active uses of the route. Connection oriented protocols normally hold on to a single

route for the duration of a connection while connectionless protocols obtain a route then discard it. The use field provides a count of the number of packets sent using that route. The interface entry indicates the network interface utilized for the route.

When *netstat* is invoked with an *interval* argument, it displays a running count of statistics related to network interfaces. This display consists of a column summarizing information for all interfaces, and a column for the interface with the most traffic since the system was last rebooted. The first line of each screen of information contains a summary since the system was last rebooted. Subsequent lines of output show values accumulated over the preceding interval.

SEE ALSO

hosts(5), networks(5), protocols(5), services(5)

BUGS

The notion of errors is ill-defined. Collisions mean something else to an IMP.

NAME

rcp – remote file copy

SYNOPSIS

rcp file1 file2
rcp [– r] file ... directory

DESCRIPTION

Rcp copies files between machines. *File* arguments may refer to either remote or local files or directories and may include absolute or relative pathnames as well. Remote files are specified in the form "rhost:file", where *rhost* is a remote hostname or alias, as in *hosts*(5). A local file name may not contain a colon (:) unless it is preceded anywhere in the name by a slash (/).

If the – r is specified and any of the source files are directories, *rcp* copies each subtree rooted at that name; in this case the destination must be a directory.

If *path* is not a full path name, it is interpreted relative to your login directory on *rhost*. A path name on a remote host may be quoted (using \, ", or ´) so that the metacharacters are interpreted remotely.

Rcp does not prompt for passwords; your current local user name must exist on *rhost* and allow remote command execution via *remsh*(1).

Rcp handles third party copies, where neither source nor target files are on the current machine. Hostnames may also take the form "rhost.rname" to use *rname* rather than the current user name on the remote host.

SEE ALSO

ftp(1), remsh(1), rlogin(1)

BUGS

Does not detect all cases where the target of a copy might be a file when only a directory should be legal.
Is confused by any output generated by commands in a *.profile* file on the remote host.

NAME

remsh – remote shell

SYNOPSIS

remsh host [– **l** username] [– **n**] command

DESCRIPTION

Remsh connects to the specified ***host***, and executes the specified ***command***. ***Remsh*** copies its standard input to the remote command, the standard output of the remote command to its standard output, and the standard error of the remote command to its standard error. Interrupt, quit and terminate signals are propagated to the remote command; ***remsh*** normally terminates when the remote command does.

The remote username used is the same as your local username, unless you specify a different remote name with the – **l** option. This remote name must be equivalent (in the sense of ***rlogin***(1)) to the originating account; no provision is made for specifying a password with a command.

If you omit ***command***, then instead of executing a single command, you will be logged in on the remote host using ***rlogin***(1).

Shell metacharacters which are not quoted are interpreted on local machine, while quoted metacharacters are interpreted on the remote machine. Thus the command

remsh otherhost cat remotefile >> localfile

appends the remote file ***remotefile*** to the localfile ***localfile***, while

remsh otherhost cat remotefile ">>" otherremotefile

appends *remotefile* to *otherremotefile*.

Host names are given in *hosts(5)*. Each host has one standard name (the first name given in the file), which is rather long and unambiguous, and optionally one or more nicknames.

SEE ALSO

rlogin(1), *hosts(5)*

BUGS

If no input is desired you should redirect the input of *remsh* to */dev/null* using the **-n** option.

You cannot run an interactive command (like *vi(1)*); use *rlogin(1)*.

NAME

rlogin – remote login

SYNOPSIS

rlogin rhost [– e*c*] [– l username] [– 8]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* to the remote host system *rhost*. Your remote terminal type is the same as your local terminal type (as given in your environment TERM variable). All echoing takes place at the remote site, so that (except for delays) the *rlogin* is transparent. Flow control via "^S" and "^Q" and flushing of input and output on interrupts are handled properly. A line of the form "~." disconnects from the remote host, where "~" is the escape character. A line of the form "~z" suspends the *rlogin* process and creates a local shell. The – 8 option allows the transmission of 8 bit data. A different escape character may be specified by the – e option. There is no space separating this option flag and the argument character.

SEE ALSO

remsh(1)

BUGS

More terminal characteristics should be propagated.

NAME

ruptime – show host status of local machines

SYNOPSIS

ruptime [– **a**] [– **l**] [– **t**] [– **u**]

DESCRIPTION

For each machine on the local network, *ruptime* prints its name, the length of time it has been up, and the average number of processes in the run queue over the last fifteen minutes. These are formed from packets broadcast by each host on the network once a minute.

Machines for which no status report has been received for five minutes are shown as being down.

Users idle an hour or more are not counted unless the – **a** flag is given.

Normally, the listing is sorted by host name. The – **l** , – **t** , and – **u** flags specify sorting by load average, uptime, and number of users, respectively.

FILES

*/usr/spool/rwho/whod.** data files

SEE ALSO

rwho(1)

NAME

rwho – who is logged in on local machines

SYNOPSIS

rwho [**-a**]

DESCRIPTION

The *rwho* command produces output similar to *who(1)*, but for all machines on the local network. If no report has been received from a machine for five minutes then *rwho* assumes the machine is down, and does not report users last known to be logged into that machine.

If a users has not typed to the system for a minute or more, then *rwho* reports this idle time. If a user has not typed to the system for an hour or more, then the user will be omitted from the output of *rwho* unless the **-a** flag is given.

FILES

*/usr/spool/rwho/whod.** information about other machines

SEE ALSO

ruptime(1), *rwhod(1m)*
who(1), UNIX System V User Reference Manual

BUGS

This is unwieldy when the number of machines on the local net is large.

NAME

telnet – user interface to the TELNET protocol

SYNOPSIS

telnet [host [port]]

DESCRIPTION

Telnet is used to communicate with another host using the TELNET protocol. If *telnet* is invoked without arguments, it enters command mode, indicated by its prompt ("telnet>"). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an *open* command (see below) with those arguments.

Once a connection has been opened, *telnet* enters input mode. In this mode, text typed is sent to the remote host. To issue *telnet* commands when in input mode, precede them with the *telnet* "escape character" which is announced at the beginning of the *telnet* session. When in command mode, the normal terminal editing conventions are available.

The following commands are available. Only enough of each command to uniquely identify it need be typed.

? [*command*]

Get help. With no arguments, *telnet* prints a help summary. If *command* is specified, *telnet* will print the help information available about the command only.

open *host* [*port*]

Open a connection to the named host. If the no port number is specified, *telnet* will attempt to contact a TELNET server at the default port. The host specification may be either a host name (see *hosts*(5)) or an Internet address

NAME

tftp – trivial file transfer program

SYNOPSIS

tftp [host] [port]

DESCRIPTION

Tftp is the user interface to the ARPANET standard Trivial File Transfer Protocol (TFTP). The program allows a user to transfer files to and from a remote network site using the User Datagram Protocol (UDP).

The client *host* with which *tftp* is to communicate may be specified on the command line. If this is done, *tftp* will immediately attempt to establish a connection to a TFTP server on that host; either way, *tftp* will enter its command interpreter and await instructions from the user. When *tftp* is awaiting commands from the user the prompt "tftp> " is provided the user. The following commands are recognized by *tftp*:

connect *host-name* [port]
Establish a connection to the supplied remote host.

get *remote-file* [*local-file*]
Retrieve *remote-file* and store it on the local machine. If the name of *local-file* is not specified, it is the same as on the remote machine. The *remote-file* may have the format *host:remote-file* at which time a connection is established to the named remote host and *remote-file* is retrieved.

mode [*mode-name*]
Set the file transfer *mode* to *mode-name*. The two possible modes are "ascii" and "binary", of which "ascii" is the default. If *mode-name* is

not specified, the current mode is printed.

put *local-file* [*remote-file*]
Store *local-file* on the remote machine. If *remote-file* is left unspecified, the name of *local-file* is used in naming the remote file. If *remote-file* has the form *host:remote-file* then a connection is established to the named host and *local-file* is sent to the remote host and named *remote-file*.

quit Terminate the TFTP session with the remote server and exit *tftp*.

status Show the current status of *tftp*.

trace Toggle packet tracing. By default, tracing is off.

verbose
Toggle *verbose* mode. If *verbose* is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, *verbose* is off.

? [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *tftp* prints a list of the known commands.

rexmt *value*
Set per-packet retransmission timeout to *value* seconds. Default *value* is five seconds.

timeout *value*
Set total retransmission timeout to *value* seconds. Default *value* is fifteen seconds.

NAME

intro – introduction to network maintenance and operation commands

DESCRIPTION

This section contains information related to network operation and maintenance. Entries whose names end in "d" are "daemons", network server programs.

LIST OF PROGRAMS

Program	*Appears on Page*	*Description*
arpbypass	arpbypass.1M	manually manipulate ARP tables
ftpd	ftpd.1M	DARPA Internet File Transfer Protocol server
gettable	gettable.1M	get NIC format host tables from a host
htable	htable.1M	convert NIC standard format host tables
ifconfig	ifconfig.1M	configure network interface parameters
rexecd	rexecd.1M	remote execution server
rlogind	rlogind.1M	remote login server
route	route.1M	manually manipulate the routing tables
routed	routed.1M	network routing daemon
rshd	rshd.1M	remote shell server
rwhod	rwhod.1M	system status server
sendmail	sendmail.1M	mail service selector
telnetd	telnetd.1M	DARPA TELNET protocol server
tftpd	tftpd.1M	DARPA Trivial File Transfer Protocol server

NAME

arpbypass – manually manipulate the ARP tables

SYNOPSIS

arpbypass *command args*

DESCRIPTION

Arpbypass is a program used to manually manipulate the network Address Resolution Protocol (ARP) tables. It is used to bypass this protocol when dealing with one or more hosts that do not use ARP. ARP is used predominately for Ethernet interfaces.

Arpbypass accepts three commands: *set*, to set an entry, *get*, to retrieve an entry; and *delete*, to remove an entry in the table.

Commands have the following syntax:

arpbypass *command* **internet-address ethernet-address flags**

where *internet-address* is the Internet address of the host we are setting (or getting or deleting) in the table, *ethernet-address* is the physical Ethernet address of the Ethernet controller in said host, and *flags* is a value specifying the type of *arp* entry to be created. The *ethernet-address* is necessary only for the *set* command.

The Internet address can be specified as a hostname or an address.

The Ethernet address is specified as an address and can take the following forms:

0xabcdef.0xghijkl, or
0xabcdef.0xgh.0xij.0xkl, or
0xab.0xcd.0xef.0xgh.0xij.0xkl

The *flags* value can be the following:

0 => entry is to be complete.
1 => entry is complete AND permanent.
2 => entry is complete AND publishable.
3 => entry is complete, permanent, AND publishable.

If no *flag* value is specified then the default value 1 is used. A publishable entry means that this host will answer the ARP request for another host on the network.

Arpbypass uses a raw socket and the SIOCSARP, SIOCGARP, and SIOCDARP *ioctl*'s to do its work. As such, only the super-user may modify the routing tables.

DIAGNOSTICS

"set internet-addr %x: ethernet-addr %x {FLAGS}"
The specified Ethernet address is being added to the tables for the host specified. The values printed are from the ARP table entry supplied in the *ioctl* call.

"get internet-addr %x: ethernet-addr %x {FLAGS}"
As above, but when looking at an entry.

"get: not in ARP table"
A get operation was attempted for an entry which wasn't present in the tables.

EXAMPLES

To add host TWG to the ARP table permanently and set its Ethernet address:

```
arpbypass set TWG 0x020701.0x002395 1
```

SEE ALSO

arp(4), route(1m)

NAME

ftpd – DARPA Internet File Transfer Protocol server

SYNOPSIS

/etc/ftpd [**–l**] [–ttimeout]

DESCRIPTION

Ftpd is the DARPA Internet File Transfer Prototocol server process. The server uses TCP and listens at the port specified in the *ftp* service specification; see *services(5)*.

If the **–l** option is specified, each *ftp* session is logged on the standard output. This allows a line of the form '/etc/ftpd -l > /tmp/ftplog" to be used to conveniently maintain a log of *ftp* sessions.

If the **–t** option is specified, an inactive session will be terminated after *timeout* seconds.

The *ftp* server currently supports the following *ftp* requests; case is not distinguished.

Request	Description
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory ("ls -lg")
MODE	specify data transfer *mode*
NLST	give name list of files in directory ("ls")
NOOP	do nothing
PASS	specify password
PORT	specify data connection port
QUIT	terminate session
RETR	retrieve a file

RNFR	specify rename-from file name
RNTO	specify rename-to file name
STOR	store a file
STRU	specify data transfer *structure*
TYPE	specify data transfer *type*
USER	specify user name
XCUP	change to parent of current working directory
XCWD	change working directory
XMKD	make a directory
XPWD	print the current working directory
XRMD	remove a directory

The remaining *ftp* requests specified in Internet RFC 765 are recognized, but not implemented.

Ftpd interprets file names according to the same "globbing" conventions used by *ftp(1)*. This allows users to utilize the metacharacters "*?[]{}~".

Ftpd authenticates users according to three rules.

1) The user name must be in the password data base, */etc/passwd*, and not have a null password. In this case a password must be provided by the client before any file operations may be performed.

2) The user name must not appear in the file *ftpusers(5)*.

3) If the user name is "ftp", an anonymous *ftp* account must be present in the password file (user "ftp"). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host's name).

In the last case, *ftpd* takes special measures to restrict the client's access privileges. The server performs a *chroot(2)* command to the home directory of the "ftp" user. In order that system security is not breached, it is recommended that the "ftp" subtree be constructed with care; the following rules are recommended.

~ftp) Make the home directory owned by "ftp" and unwritable by anyone.

~ftp/bin)
Make this directory owned by the super-user and unwritable by anyone. The program *ls*(1) must be present to support the list commands. This program should have mode 111.

~ftp/etc)
Make this directory owned by the super-user and unwritable by anyone. The files *passwd(5)* and *group(5)* must be present for the *ls* command to work properly. These files should be mode 444.

~ftp/pub)
Make this directory mode 777 and owned by "ftp". Users should then place files which are to be accessible via the anonymous account in this directory.

SEE ALSO

ftp(1)
chroot(2), UNIX System V Programmer Reference Manual

BUGS

There is no support for aborting commands.

The anonymous account is inherently dangerous and should avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user id of the logged in user and uses non-privileged port numbers for the data connection.

NAME

gettable – get NIC format host tables from a host

SYNOPSIS

/etc/gettable *host* [*file*]

DESCRIPTION

Gettable is a simple program used to obtain the NIC standard host tables from a "nicname" server. The indicated *host* is queried for the tables. The tables, if retrieved, are placed in the file *hosts.txt* unless *file* is specified, in which case the table is placed in *file*.

Gettable operates by opening a TCP connection to the port indicated in the service specification for "nicname". A request is then made for "ALL" names and the resultant information is placed in the output file.

Gettable is best used in conjunction with the *htable(1m)* program which converts the NIC standard file format to that used by the network library lookup routines.

SEE ALSO

intro(3N), htable(1M)

BUGS

Should allow requests for only part of the database.

NAME

htable – convert NIC standard format host tables

SYNOPSIS

/etc/htable [–c connected-nets] [–l local-nets] [input-file]

DESCRIPTION

Htable is used to convert host files in the format specified in Internet RFC 810 to the format used by the network library routines. Three files are created as a result of running *htable*: *hosts*, *networks*, and *gateways*. The *hosts* file is used by the *gethostent(3N)* routines in mapping host names to addresses. The *networks* file is used by the *getnetent(3N)* routines in mapping network names to numbers. The *gateways* file is used by the routing daemon in identifying "passive" Internet gateways; see *routed(1M)* for an explanation.

If any of the files *localhosts*, *localnetworks*, or *localgateways* are present in the current directory, the file's contents is prepended to the output file without interpretation. This allows sites to maintain local aliases and entries which are not normally present in the master database.

Connected-nets is a comma-separated list of networks that your local host is connected to. This list is used in generating the *gateways* file. *Local-nets* is a comma-separated list of networks. Hosts that are on these networks are not included in the *hosts* file. This prevents duplicate entries in the *hosts* file when the *localhosts* file is prepended to it. If *input-file* is not specified, *htable* reads the input from "stdin".

Htable is best used in conjunction with the *gettable(1M)* program which retrieves the NIC database from a host.

SEE ALSO

intro(3N), gettable(1M)

BUGS

Does not properly calculate the *gateways* file.

NAME

ifconfig – configure network interface parameters

SYNOPSIS

/etc/ifconfig interface [*address* [*destination*]] [*parameters*]

DESCRIPTION

Ifconfig is used to assign an address to a network interface and/or configure network interface parameters. *Ifconfig* must be used at boot time to define the network address of each interface present on a machine; it may also be used at a later time to redefine an interface's address. The *interface* parameter is a string of the form "name unit", e.g., "en0", while the *address* is either a host name present in the host name data base, *hosts(5)*, or a DARPA Internet address expressed in the Internet standard "dot notation". *Destination* is the address of the other end of a point-to-point network.

The following parameters may be set with *ifconfig*:

up Mark an interface "up".

down Mark an interface "down". When an interface is marked "down", the system will not attempt to transmit messages through that interface.

trailers Enable the use of a "trailer" link level encapsulation when sending (default). If a network interface supports *trailers*, the system will, when possible, encapsulate outgoing messages in a manner which minimizes the number of memory to memory copy operations performed by the receiver.

-trailers Disable the use of a "trailer" link level encapsulation.

arp Enable the use of the Address Resolution Protocol in mapping between network level addresses and link level addresses (default). This is currently implemented for mapping between DARPA Internet addresses and 10Mb/s Ethernet addresses.

-arp Disable the use of the Address Resolution Protocol.

Ifconfig displays the current configuration for a network interface when no optional parameters are supplied.

Only the super-user may modify the configuration of a network interface.

DIAGNOSTICS

Messages indicating the specified interface does not exist, the requested address is unknown, the user is not privileged and tried to alter an interface's configuration.

SEE ALSO

intro(4N), netstat(1)

NAME

rexecd – remote execution server

SYNOPSIS

/etc/rexecd

DESCRIPTION

Rexecd is the server for the *rexec(3)* routine. The server provides remote execution facilities with authentication based on user names and encrypted passwords.

Rexecd listens for service requests at the port indicated in the "exec" service specification; see *services(5)*. When a service request is received the following protocol is initiated:

1) The server reads characters from the socket or transport end point up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

2) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine.

3) A null terminated user name of at most 16 characters is retrieved on the initial socket.

4) A null terminated, encrypted, password of at most 16 characters is retrieved on the initial socket.

5) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

6) *Rexecd* then validates the user as is done at login time and, if the authentication was successful, changes to the user's home directory, and establishes the user and group protections of the user. If any of these steps fail the connection is aborted with a diagnostic message returned.

7) A null byte is returned on the connection associated with the **stderr** and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rexecd*.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 7 above upon successful completion of all the steps prior to the command execution).

"username too long"
The name is longer than 16 characters.

"password too long"
The password is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Login incorrect."
No password file entry for the user name existed.

"No remote directory."
The *chdir* command to the home directory failed.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

SEE ALSO

intro(3W)

BUGS

A facility to allow all data exchanges to be encrypted should be present.

NAME

rlogind – remote login server

SYNOPSIS

/etc/rlogind

DESCRIPTION

Rlogind is the server for the *rlogin(1)* program. The server provides a remote login facility with authentication based on privileged port numbers.

Rlogind listens for service requests at the port indicated in the "login" service specification; see *services(5)*. When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts(5)*), the server aborts the connection.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal (see *pty(4)*), and manipulates file descriptors so that the slave half of the pseudo terminal becomes the **stdin** , **stdout** , and **stderr** for a login process. The login process is an instance of the *login(1)* program, invoked with the **–r** option. The login process then proceeds with the authentication process as described in *rshd(1m)*, but if automatic authentication fails, it reprompts the user to login as one finds on a standard terminal line.

The parent of the login process manipulates the master side of the pseudo terminal, operating as an intermediary between the login process and the client instance of the *rlogin* program. In normal operation, the packet protocol described in *pty(4)* is invoked to provide ^S/^Q type facilities and propagate interrupt signals to the remote programs. The login process propagates the client terminal's baud rate and terminal type, as found in the shell variable, "TERM".

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1.

"Hostname for your address unknown."
No entry in the host name database existed for the client's machine.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

NAME

route – manually manipulate the routing tables

SYNOPSIS

/etc/route [**–f**] [*command args*]

DESCRIPTION

Route is a program used to manually manipulate the network routing tables. It normally is not needed, as the system routing table management daemon, *routed(1M)*, should tend to this task.

Route accepts three commands: *add*, to add a route; *delete*, to delete a route; and *change*, to modify an existing route.

All commands have the following syntax:

/etc/route *command* **destination gateway** [**metric**]

where *destination* is a host or network for which the route is "to", *gateway* is the gateway to which packets should be addressed, and *metric* is an optional count indicating the number of hops to the *destination*. Routes to a particular host are distinguished from those to a network by interpreting the Internet address associated with *destination*. If the *destination* has a "local address part" of INADDR_ANY, then the route is assumed to e to a network; otherwise, it is presumed to be a route to a host. If the route is to a destination connected via a gateway, the *metric* should be greater than "0". All symbolic names specified for a *destination* or *gateway* are looked up first in the host name database, *hosts(5)*. If this lookup fails, the name is then looked for in the network name database, *networks(5)*.

Route uses a raw socket and the SIOCADDRT and SIOCDELRT *ioctl*'s to do its work. As such, only the superuser may modify the routing tables.

If the **-f** option is specified, *route* will "flush" the routing tables of all gateway entries. If this is used in conjunction with one of the commands described above, the tables are flushed prior to the command's application.

DIAGNOSTICS

"add %s: gateway %s flags %x"
The specified route is being added to the tables. The values printed are from the routing table entry supplied in the *ioctl* call.

"delete %s: gateway %s flags %x"
As above, but when deleting an entry.

"%s %s done"
When the **-f** flag is specified, each routing table entry deleted is indicated with a message of this form.

"not in table"
A delete operation was attempted for an entry which wasn't present in the tables.

"routing table overflow"
An add operation was attempted, but the system was low on resources and was unable to allocate memory to create the new entry.

SEE ALSO

intro(4), routed(1M)

BUGS

The change operation is not implemented, one should add the new route, then delete the old one.

NAME

routed – network routing daemon

SYNOPSIS

/etc/routed [**–s**] [**–q**] [**–t**] [*logfile*]

DESCRIPTION

Routed is invoked at boot time to manage the network routing tables. The routing daemon uses a variant of the Xerox NS Routing Information Protocol in maintaining up to date kernel routing table entries.

In normal operation *routed* listens on *udp(4)* socket 520 (decimal) for routing information packets. If the host is an internetwork router, it periodically supplies copies of its routing tables to any directly connected hosts and networks.

When *routed* is started, it uses the SIOCGIFCONF *ioctl* to find those directly connected interfaces configured into the system and marked "up" (the software loopback interface is ignored). If multiple interfaces are present, it is assumed the host will forward packets between networks. *Routed* then transmits a *request* packet on each interface (using a broadcast packet if the interface supports it) and enters a loop, listening for *request* and *response* packets from other hosts.

When a *request* packet is received, *routed* formulates a reply based on the information maintained in its internal tables. The *response* packet generated contains a list of known routes, each marked with a "hop count" *metric* (a count of 16, or greater, is considered "infinite"). The *metric* associated with each route returned provides a *metric relative to the sender*.

Response packets received by *routed* are used to update the routing tables if one of the following conditions is satisfied:

(1) No routing table entry exists for the destination network or host, and the *metric* indicates the destination is "reachable" (i.e. the hop count is not infinite).

(2) The source host of the packet is the same as the router in the existing routing table entry. That is, updated information is being received from the very internetwork router through which packets for the destination are being routed.

(3) The existing entry in the routing table has not been updated for some time (defined to be 90 seconds) and the route is at least as cost effective as the current route.

(4) The new route describes a shorter route to the destination than the one currently stored in the routing tables; the *metric* of the new route is compared against the one stored in the table to decide this.

When an update is applied, *routed* records the change in its internal tables and generates a *response* packet to all directly connected hosts and networks. *Routed* waits a short period of time (no more than 30 seconds) before modifying the kernel's routing tables to allow possible unstable situations to settle.

In addition to processing incoming packets, *routed* also periodically checks the routing table entries. If an entry has not been updated for three minutes, the entry's *metric* is set to infinity and marked for deletion.

Deletions are delayed an additional 60 seconds to insure the invalidation is propagated throughout the Internet.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks.

Supplying the **–s** option forces *routed* to supply routing information whether it is acting as an internetwork router or not. The **–q** option is the opposite of the **–s** option. If the **–t** option is specified, all packets sent or received are printed on the standard output. In addition, *routed* will not divorce itself from the controlling terminal so that interrupts from the keyboard will kill the process. Any other argument supplied is interpreted as the name of file in which *routed*'s actions should be logged. This log contains information about any changes to the routing tables and a history of recent messages sent and received which are related to the changed route.

In addition to the facilities described above, *routed* supports the notion of "distant" *passive* and *active gateways*. When *routed* is started up, it reads the file */etc/gateways* to find *gateways* which may not be identified using the SIOGIFCONF *ioctl*. *Gateways* specified in this manner should be marked passive if they are not expected to exchange routing information, while *gateways* marked active should be willing to exchange routing information (i.e. they should have a *routed* process running on the machine). Passive *gateways* are maintained in the routing tables forever and information regarding their existence is included in any routing information transmitted. Active *gateways* are treated equally to network interfaces. Routing

information is distributed to the *gateway* and if no routing information is received for a period of the time, the associated route is deleted.

FILES

/etc/gateways for distant gateways

SEE ALSO

"Internet Transport Protocols", XSIS 028112, Xerox System Integration Standard.
udp(4)

BUGS

The kernel's routing tables may not correspond to those of *routed* for short periods of time while processes utilizing existing routes exit; the only remedy for this is to place the routing process in the kernel.

Routed should listen to intelligent interfaces, such as an IMP, and to error protocols, such as ICMP, to gather more information.

NAME

rshd – remote shell server

SYNOPSIS

/etc/rshd

DESCRIPTION

Rshd is the server for the *rcmd(3)* routine and, consequently, for the *remsh(1)* program. The server provides remote execution facilities with authentication based on privileged port numbers.

Rshd listens for service requests at the port indicated in the "cmd" service specification; see *services(5)*. When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

3) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine. The source port of this second connection is also in the range 0-1023.

4) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts(5)*), the server aborts the connection.

5) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as a user identity to use on the *server*'s machine.

6) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as the user identity on the *client*'s machine.

7) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

8) *Rshd* then validates the user according to the following steps. The remote user name is looked up in the password file and a *chdir* is performed to the user's home directory. If either the lookup or *chdir* fail, the connection is terminated. If the user is not the superuser, (user id 0), the file */etc/hosts.equiv* is consulted for a list of hosts considered "equivalent". If the client's host name is present in this file, the authentication is considered successful. If the lookup fails, or the user is the superuser, then the file *.rhosts* in the home directory of the remote user is checked for the machine name and identity of the user on the client's machine. If this lookup fails, the connection is terminated.

9) A null byte is returned on the connection associated with the **stderr** and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rshd*.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of "1" ("0" is returned in step 9 above upon successful completion of all the steps prior to the command execution).

"locuser too long"
The name of the user on the client's machine is longer than 16 characters.

"remuser too long"
The name of the user on the remote machine is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Hostname for your address unknown."
No entry in the host name database existed for the client's machine.

"Login incorrect."
No password file entry for the user name existed.

"No remote directory."
The *chdir* command to the home directory failed.

"Permission denied."
The authentication procedure described above failed.

"Can't make pipe."
The pipe needed for the **stderr**, wasn't created.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

SEE ALSO

remsh(1), rcmd(3)

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

NAME

rwhod – system status server

SYNOPSIS

/etc/rwhod

DESCRIPTION

Rwhod is the server which maintains the database used by the *rwho(1)* and *ruptime(1)* programs. Its operation is predicated on the ability to *broadcast* messages on a network.

Rwhod operates as both a producer and consumer of status information. As a producer of information it periodically queries the state of the system and constructs status messages which are broadcast on a network. As a consumer of information, it listens for other *rwhod* servers' status messages, validating them, then recording them in a collection of files located in the directory */usr/spool/rwho*.

The *rwho* server transmits and receives messages at the port indicated in the "rwho" service specification, see *services(5)*. The messages sent and received, are of the form:

```
struct  outmp {
        char    out_line[8];/* tty name */
        char    out_name[8];/* user id */
        long    out_time;/* time on */
};

struct  whod {
        char    wd_vers;
        char    wd_type;
        char    wd_fill[2];
        int     wd_sendtime;
        int     wd_recvtime;
```

```
        char    wd_hostname[32];
        int     wd_loadav[3];
        int     wd_boottime;
        struct  whoent {
                struct  outmp we_utmp;
                int     we_idle;
        } wd_we[1024 / sizeof (struct whoent)];
};
```

All fields are converted to network byte order prior to transmission. The load averages are the load averages over the 5, 10, and 15 minute intervals prior to a server's transmission. The host name included is that returned by *gethostname(3W)*. The array at the end of the message contains information about the users logged in to the sending machine. This information includes the *ttyname*, *userid*, *hostname* (if remote), and login time for each non-idle terminal line and a value indicating the time since a character was last received on the terminal line.

Messages received by the *rwho* server are discarded unless they originated at an *rwho* server's port. In addition, if the host's name, as specified in the message, contains any unprintable ASCII characters, the message is discarded. Valid messages received by *rwhod* are placed in files named *whod.hostname* in the directory */usr/spool/rwho*. These files contain only the most recent message, in the format described above.

Status messages are generated approximately once every 60 seconds. *Rwhod* performs an *nlist(3)* on */unix* every ten minutes to guard against the possibility that this file is not the system image currently operating.

SEE ALSO

rwho(1), ruptime(1)
nlist(3), **UNIX System V Programmer Reference Manual**

BUGS

Should relay status information between networks. People often interpret the server dying as a machine going down.

NAME

sendmail – send mail over the Internet

SYNOPSIS

/usr/lib/sendmail [flags] [address ...]

newaliases

mailq

DESCRIPTION

Sendmail sends a message to one or more people, routing the message over whatever networks are necessary. *Sendmail* does internetwork forwarding as necessary to deliver the message to the correct place.

Sendmail is not intended as a user interface routine; other programs provide user-friendly front ends; *sendmail* is used only to deliver pre-formatted messages.

With no flags, *sendmail* reads its standard input up to a control-D or a line with a single dot and sends a copy of the letter found there to all of the addresses listed. It determines the network to use based on the syntax and contents of the addresses.

Local addresses are looked up in a file and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash. Normally the sender is not included in any alias expansions, e.g., if 'john' sends to 'group', and 'group' includes 'john' in the expansion, then the letter will not be delivered to 'john'.

Flags are:

–ba Go into ARPANET mode. All input lines must end with a CR-LF, and all messages will be generated with a CR-LF at the end. Also, the

	"From:" and "Sender:" fields are examined for the name of the sender.
-bd	Run as a daemon.
-bi	Initialize the alias database.
-bm	Deliver mail in the usual way (default).
-bp	Print a listing of the queue.
-bs	Use the SMTP protocol as described in RFC821. This flag implies all the operations of the **-ba** flag that are compatible with SMTP.
-bt	Run in address test mode. This mode reads addresses and shows the steps in parsing; it is used for debugging configuration tables.
-bv	Verify names only; do not try to collect or deliver a message. This flag is currently non-functional: it reports all mail as deliverable.
-bz	Create the configuration freeze file.
-C*file*	Use alternate configuration file.
-d*X*	Set debugging value to *X*.
-F*fullname*	Set the full name of the sender.
-f*name*	Sets the name of the "from" person (i.e., the sender of the mail). **-f** can only be used by the special users *root*, *daemon*, and *network*, or if the person you are trying to become is the same as the person you are.

-h*N*	Set the hop count to *N*. The hop count is incremented every time the mail is processed. When it reaches a limit, the mail is returned with an error message, the victim of an aliasing loop.
-n	Do not do aliasing.
-o*x value*	Set option *x* to the specified *value*. Options are described below.
-q[*time*]	Process saved messages in the queue at given intervals. If *time* is omitted, process the queue once. *Time* is given as a tagged number, with "s" being seconds, "m" being minutes, "h" being hours, "d" being days, and "w" being weeks. For example, "-q1h30m" or "-q90m" would both set the *timeout* to one hour thirty minutes.
-r*name*	An alternate and obsolete form of the -f flag.
-t	Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for people to send to. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed.
-v	Go into verbose mode. Alias expansions will be announced, etc.

There are also a number of processing options that may be set. Normally these will only be used by a system administrator. Options may be set either on the

command line using the –o flag or in the configuration file. The options are:

Afile	Use alternate alias file.
c	On mailers that are considered "expensive" to connect to, don't initiate immediate connection. This requires queueing.
d*x*	Set the delivery mode to *x*. Delivery modes are "i" for interactive (synchronous) delivery, "b" for background (asynchronous) delivery, and "q" for queue only, i.e., actual delivery is done the next time the queue is run.
D	Try to automatically rebuild the alias database if necessary.
e*x*	Set error processing to mode *x*. Valid modes are "m" to mail back the error message, "w" to "write" back the error message (or mail it back if the sender is not logged in), "p" to print the errors on the terminal (default), and "q" to throw away error messages (only exit status is returned). If the text of the message is not mailed back by modes "m" or "w" and if the sender is local to this machine, a copy of the message is appended to the file "dead.letter" in the sender's home directory.
F*mode*	The mode to use when creating

	temporary files.
f	Save UNIX-style From lines at the front of messages.
g*N*	The default group *id* to use when calling mailers.
H*file*	The SMTP help file.
i	Do not take dots on a line by themselves as a message terminator.
L*n*	The log level.
m	Send to "me" (the sender) also if I am in an alias expansion.
o	If set, this message may have old style headers. If not set, this message is guaranteed to have new style headers (i.e., commas instead of spaces between addresses). If set, an adaptive algorithm is used that will correctly determine the header format in most cases.
Q*queuedir*	Select the directory in which to queue messages.
r*timeout*	The *timeout* on reads; if none is set, *sendmail* will wait forever for a mailer.
S*file*	Save statistics in the named file.
s	Always instantiate the queue file, even under circumstances where it is not strictly necessary.
T*time*	Set the *timeout* on messages in the queue to the specified time. After

sitting in the queue for this amount of time, they will be returned to the sender. The default is three days.

t*stz,dtz* Set the name of the time zone.

u*N* Set the default user *id* for mailers.

If the first character of the user name is a vertical bar, the rest of the user name is used as the name of a program to pipe the mail to. It may be necessary to quote the name of the user to keep *sendmail* from suppressing the blanks from between arguments.

Sendmail returns an exit status describing what it did. The codes are defined in <*sysexits.h*>

EX_OK	Successful completion on all addresses.
EX_NOUSER	User name not recognized.
EX_UNAVAILABLE	Catchall meaning necessary resources were not available.
EX_SYNTAX	Syntax error in address.
EX_SOFTWARE	Internal software error, including bad arguments.
EX_OSERR	Temporary operating system error, such as "cannot fork".
EX_NOHOST	Host name not recognized.
EX_TEMPFAIL	Message could not be sent immediately, but was queued.

If invoked as *newaliases*, *sendmail* will rebuild the alias database. If invoked as *mailq*, *sendmail* will print the contents of the mail queue.

FILES

Except for */usr/lib/sendmail.cf*, these pathnames are all specified in */usr/lib/sendmail.cf*. Thus, these values are only approximations.

/usr/lib/aliases	raw data for alias names
/usr/lib/aliases.pag	
/usr/lib/aliases.dir	data base of alias names
/usr/lib/sendmail.cf	configuration file
/usr/lib/sendmail.fc	frozen configuration
/usr/lib/sendmail.hf	help file
/usr/lib/sendmail.st	collected statistics
/usr/bin/uux	to deliver uucp mail
/usr/net/bin/v6mail	to deliver local mail
/usr/net/bin/sendberkmail	to deliver Berknet mail
/usr/lib/mailers/arpa	to deliver ARPANET mail
/usr/spool/mqueue/*	temp files

SEE ALSO

mailx(1),
DARPA Internet Request For Comments RFC819, RFC821, RFC822;

BUGS

Sendmail converts blanks in addresses to dots. This is incorrect according to the old ARPANET mail protocol RFC733 (NIC 41952), but is consistent with the new protocols (RFC822).

NAME

telnetd – DARPA TELNET protocol server

SYNOPSIS

/etc/telnetd [**–d**] [*port*]

DESCRIPTION

Telnetd is a server which supports the DARPA standard TELNET virtual terminal protocol. The TELNET server operates at the port indicated in the "telnet" service description; see *services(5)*. This port number may be overridden (for debugging purposes) by specifying a port number on the command line. If the **–d** option is specified, each socket created by *telnetd* will have debugging enabled (see SO_DEBUG in *socket(3W)*).

Telnetd operates by allocating a pseudo-terminal device (see *pty(4)*) for a client, then creating a login process which has the slave side of the pseudo-terminal as **stdin**, **stdout**, and **stderr**. *Telnetd* manipulates the master side of the pseudo terminal, implementing the TELNET protocol and passing characters between the client and login process.

When a TELNET session is started up, *telnetd* sends a TELNET option to the client side indicating a willingness to do "remote echo" of characters. The pseudo terminal allocated to the client is configured to operate in "cooked" mode, and with XTABS and CRMOD enabled (see *tty(4)* in the UNIX System V Progammer Reference Manual). Aside from this initial setup, the only mode changes *telnetd* will carry out are those required for echoing characters at the client side of the connection.

NAME

tftpd – DARPA Trivial File Transfer Protocol server

SYNOPSIS

/etc/tftpd [*port*]

DESCRIPTION

Tftpd is a server which supports the DARPA Trivial File Transfer Protocol. The TFTP server operates at the port indicated in the "tftp" service description; see *services(5)* . This port number may be overridden (for debugging purposes) by specifying a port number on the command line.

The use of *tftp* does not require an account or password on the remote system. Due to the lack of authentication information, *tftpd* will allow only publicly readable files to be accessed. Note that this extends the concept of "public" to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling *tftp* service.

SEE ALSO

tftp(1)

BUGS

This server is known only to be self consistent (i.e. it operates with the user TFTP program, *tftp(1)*). Due to the unreliability of the transport protocol (UDP) and the scarcity of TFTP implementations, it is uncertain whether it really works.

The search permissions of the directories leading to the files accessed are not checked.

NAME

intro – introduction to network library functions

DESCRIPTION

This section describes the functions that may be found in various libraries:

(3) These are functions used in network programming which do not fall conveniently into any larger grouping.

(3N) These functions constitute the DARPA internet network library. They may be used in conjunction with the functions in Section (3T) or (3W).

(3T) These functions are an implementation of the user-level portion of the AT&T Transport Level Interface. They may be used by networking application programs to solicit the services of the Transport Level Protocols *tcp(4)* and *udp(4)*.

(3W) These functions constitute the Wollongong Socket Compatibility Library. They comprise a complete and coherent interface to the transport level protocols *tcp(4)* and *udp(4)*. They are fully compatible with the corresponding functions in the 4.2 BSD release of the UNIX operating system. They may be used by networking application programs to solicit transport level services. When using /usr/ethernet/include/* files with a non-filename compiler, use "_DSHORT_IDENT" and "_DSYSTEMS" in the compile line.

SEE ALSO

intro(3N), intro(3T), intro(3W)

NAME

bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

```
bcopy(b1, b2, length)
char *b1, *b2;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(l)
int l;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string*(3) do.

Bcopy copies *length* bytes from string *b1* to the string *b2*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

Ffs finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at "1". A return value of "0" indicates the value passed is zero.

BUGS

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy*.

SEE ALSO

string(3), **UNIX System V Programmer Reference Manual**

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
u_short inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;
```

DESCRIPTION

Rcmd is a routine used by the superuser to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *Rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *Ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd(1M)* server (among others).

Rcmd looks up the host **ahost* using *gethostbyname*(3N), returning "– 1" if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is zero, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in *rshd(1M)*.

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the superuser is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a *gethostent*(3N) routine, two user names and a flag indicating if the local user's name is the superuser. It then checks *hosts.equiv(5)* and, possibly, *.rhosts(5)* in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A "1" is returned if the machine name is listed in *hosts.equiv(5)* or the host and remote user name are found in *.rhosts(5)*; otherwise *ruserok* returns "0". If the *superuser* flag is "1", the checking of *host.equiv(5)* is bypassed.

SEE ALSO

remsh(1), *rlogin(1)*, *rexec(3)*, *gethostbyname(3N)*, *gethostent(3N)*, *hosts.equiv(5)*, *.rhosts(5)*, *rexecd(1M)*, *rlogind(1M)*, *rshd(1M)*

BUGS

There is no way to specify options to the *socket* call which *rcmd* makes.

NAME

rexec – return stream to a remote command

SYNOPSIS

```
rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
u_short inport;
char *user, *passwd, *cmd;
int *fd2p;
```

DESCRIPTION

Rexec looks up the host **ahost* using *gethostbyname*(3N), returning "– 1" if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call "getservbyname("exec", "tcp")" (see *getservent*(3N)). The protocol for connection is described in detail in *rexecd(1M)*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then a auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If

fd2p is zero, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3), gethostbyname(3N), getservbyname(3N), .netrc(5), hosts(5), rexecd(1M)

BUGS

There is no way to specify options to the *socket* call which *rexec* makes.

NAME

intro – introduction to network library functions

DESCRIPTION

This section describes functions that are applicable to the DARPA Internet network.

LIST OF FUNCTIONS

Name	*On Page*	*Description*
endhostent	gethostent.3n	get network host entry
endnetent	getnetent.3n	get network entry
endprotoent	getprotoent.3n	get protocol entry
endservent	getservent.3n	get service entry
gethostbyaddr	gethostent.3n	get network host entry
gethostbyname	gethostent.3n	get network host entry
gethostent	gethostent.3n	get network host entry
getnetbyaddr	getnetent.3n	get network entry
getnetbyname	getnetent.3n	get network entry
getnetent	getnetent.3n	get network entry
getprotobyname	getprotoent.3n	get protocol entry
getprotobynumber	getprotoent.3n	get protocol entry
getprotoent	getprotoent.3n	get protocol entry
getservbyname	getservent.3n	get service entry
getservbyport	getservent.3n	get service entry
getservent	getservent.3n	get service entry
htonl	byteorder.3n	convert values between host and network byte order
htons	byteorder.3n	convert values

		between host and network byte order
inet_addr	inet.3n	Internet address manipulation routines
inet_lnaof	inet.3n	Internet address manipulation routines
inet_makeaddr	inet.3n	Internet address manipulation routines
inet_netof	inet.3n	Internet address manipulation routines
inet_network	inet.3n	Internet address manipulation routines
ntohl	byteorder.3n	convert values between host and network byte order
ntohs	byteorder.3n	convert values between host and network byte order
sethostent	gethostent.3n	get network host entry
setnetent	getnetent.3n	get network entry
setprotoent	getprotoent.3n	get protocol entry
setservent	getservent.3n	get service entry

NAME

htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. They are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostent*(3N) and *getservent*(3N).

SEE ALSO

gethostent(3N), getservent(3N)

NAME

gethostent, gethostbyaddr, gethostbyname, sethostent, endhostent – get network host entry

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

sethostent(stayopen)
int stayopen

endhostent()
```

DESCRIPTION

Gethostent, *gethostbyname*, and *gethostbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network host data base, */etc/hosts*.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned;

currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr A pointer to the network address for the host. Host addresses are returned in network byte order.

Gethostent reads the next line of the file, opening the file if necessary.

Sethostent opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base will not be closed after each call to *gethostent* (either directly, or indirectly through one of the other "gethost" calls).

Endhostent closes the file.

Gethostbyname and *gethostbyaddr* sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network order.

SEE ALSO

hosts(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
int net, type;

setnetent(stayopen)
int stayopen

endnetent()
```

DESCRIPTION

Getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, */etc/networks*.

```
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net number type */
        long    n_net;          /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

Getnetent reads the next line of the file, opening the file if necessary.

Setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the *networks(5)* data base will not be closed after each call to *getnetent* (either directly, or indirectly through one of the other "getnet" calls).

Endnetent closes the file.

Getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching network name or network address is found, or until EOF is encountered. Network numbers are supplied in host order. *Type* is the type of the network (e.g. AF_INET).

SEE ALSO

networks(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen

endprotoent()
```

DESCRIPTION

Getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

```
struct  protoent {
 char   *p_name;      /* official name of protocol */
 char   **p_aliases;  /* alias list */
 long   p_proto;      /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

Getprotoent reads the next line of the file, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the *protocols(5)* data base will not be closed after each call to *getprotoent* (either directly, or indirectly through one of the other "getproto" calls).

Endprotoent closes the file.

Getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

SEE ALSO

protocols(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen

endservent()
```

DESCRIPTION

Getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct  servent {
 char   *s_name;        /* official name of service */
 char   **s_aliases;    /* alias list */
 long   s_port;         /* port service resides at */
 char   *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in

network byte order.

s_proto The name of the protocol to use when contacting the service.

Getservent reads the next line of the file, opening the file if necessary.

Setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the *services(5)* data base will not be closed after each call to *getservent* (either directly, or indirectly through one of the other "getserv" calls).

Endservent closes the file.

Getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching service name or port number is found, or until EOF is encountered. If a service name is also supplied (non-NULL), searches must also match the protocol.

SEE ALSO

getprotoent(3N), services(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

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

struct in_addr inet_addr(cp)
char *cp;

int inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in "." notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof*

break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the "." notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "." notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e. a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostent(3N), getnetent(3N), hosts(5), networks(5)

DIAGNOSTICS

The value – 1 is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed. The string returned by *inet_ntoa* resides in a static memory area.

The routine *inet_makeaddr* is currently only capable of forming Class A addresses and its first argument *net*, must therefore be less than 128.

NAME

intro – introduction to Transport Level Interface Library

SYNOPSIS

#include <errno.h>
#include <sys/tli.h>

DESCRIPTION

This section describes the functions that make up the Transport Level Interface Library. These functions constitute a self-contained interface to the transport level protocols, *tcp(4)* and *udp(4)*. They may be used in conjunction with the functions in Section (3N), but not with those in Section (3W).

Transport Level Interface functions enable a transport user (such as the session layer or a networking application) to initiate requests to the transport provider and process incoming events.

Two modes of service, *connection mode* and *connectionless mode*, are provided by the Transport Interface. Connection mode is circuit-oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner. It also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless mode, in contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. This service requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted. All the

information required to deliver a unit of data (for example, the destination address) is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained. Connectionless-mode service is attractive for applications that:

- involve short-term request/response interactions,
- exhibit a high level of redundancy,
- are dynamically reconfigurable,
- or do not require guaranteed, in-sequence delivery of data.

DIAGNOSTICS

Most of these calls have one or more error returns. An error condition is indicated bya return value of "-1"; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable *t_errno*, which is not cleared on successful calls. Thus *t_errno* should be tested only after an error has occurred.

The following is a complete list of the errors and their names.

01 TBADF Invalid file descriptor

The specified file descriptor does not refer to a transport endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication

arrived.

02 TBADOPT Invalid protocol options
The specified protocol options were in an incorrect format or contained illegal information.

03 TBADFLAG Invalid flag
An invalid flag was specified.

04 TBADADDR Invalid address
The specified protocol address was in an incorrect format or contained illegal information.

06 TBADSEQ Invalid sequence
An invalid sequence number was specified.

07 TBADDATA Invalid data

08 TOUTSTATE
The primitive was issued in the wrong sequence.

09 TACCES Permission denied
The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.

10 TFLOW
O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

11 TBUFOVFLW Buffer overflow
The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.

12 TSTATECHNG State change
The transport provider is undergoing a state change.

13 TNOADDR No address
The transport provider could not allocate an address.

14 TNODATA No data currently available
O_NDELAY was set, but no connect indications had been queued.

15 TNODIS No disconnect indication
No disconnect indication currently exists on the specified transport endpoint.

16 TNOREL
No orderly release indication currently exists on the specified transport endpoint.

17 TNOUDERR
No unit data error indication currently exists on the specified transport endpoint. No orderly release indication currently exists on the specified transport endpoint.

18 TNOTSUPPORT
This primitive is not supported by the underlying transport provider.

20 TSYSERR
A system error has occurred during execution of this primitive. The provider's state may change to T_DATAXFER. Check the *err_no* variable to determine the error code of the system error.

SEE ALSO

The Network Programmer's Guide for the AT&T 3B2

Computer, Unix System V Release 3.

NAME

t_accept – accept a connect request

SYNOPSIS

```
#include <sys/tli.h>
int t_accept(fd, resfd, call)
int fd;
int resfd;
struct t_call *call;
```

DESCRIPTION

This primitive is issued by a passive transport user to accept a *t_connect(3T)* request. *Fd* identifies the local transport endpoint where the connect indication arrived, *resfd* specifies the local transport endpoint where the connection is to be established, and *call* specifies the call structure to be sent to the calling transport user.

In the *call* structure, *addr* is the address of the caller, *opt* and *udata* are unused, and *sequence* is a number that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on a different local transport endpoint than the one on which the connect indication arrived. This alternate transport endpoint is specified by *resfd*, where the endpoint must be bound to a protocol address and must be in the idle state before the *t_accept* is issued. It is legal to accept a connection on the same transport endpoint on which the connect indication arrived (i.e. fd == resfd). However, the connect indication specified by *sequence* (see description below) must be the only outstanding connect indication associated with *fd*, otherwise the routine will fail.

The value of *sequence* is used by the passive transport user to support multi-threading of connect indications/responses. It enables the user to listen for multiple connect indications (via *t_listen(3T)* before responding to any of them. When the user eventually responds, *sequence* tells the transport provider to which connect indication the response is associated. A transport user may not accept a connect request until it receives all connect indications and disconnect indications that have arrived on the transport endpoint; any attempt to do so will fail and set *t_errno* to TLOOK.

DIAGNOSTICS

t_accept returns 0 on success and -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived.

TOUTSTATE
: The primitive was issued in the wrong sequence.

TACCES
: The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.

TBADSEQ
: An invalid sequence number was specified.

TLOOK
: An asynchronous event has occurred on this transport endpoint and requires immediate

attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive. The provider's state may change to T_DATAXFER.

NAME

t_alloc – allocate a library structure

SYNOPSIS

```
#include <sys/tli.h>
char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

DESCRIPTION

The *t_alloc* routine dynamically allocates memory for the various transport library argument structures. The routine will allocate memory for the specified structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by *struct_type*, and can be one of the following:

T_BIND
: struct t_bind

T_CALL
: struct t_call

T_OPTMGMT
: struct t_optmgmt

T_DIS
: struct t_discon

T_UNITDATA
: struct t_unitdata

T_UDERROR
: struct t_uderr

T_INFO
: struct t_info

where each of these structures is used as an

argument to one or more transport primitives.

Each of the above structures, except T_INFO , contains at least one field of type *struct netbuf*. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The *fields* argument specifies this option, where the argument is the bitwise-OR of any of the following:

T_ADDR
: the *addr* field of the *t_bind*, *t_call*, *t_unitdata*, or *t_uderr* structures.

T_OPT
: the *opt* field of the *t_optmgmt*, *t_call*, *t_unitdata*, or *t_uderr* structures.

T_UDATA
: the *udata* field of the *t_call*, *t_discon*, or *t_unitdata* structures.

T_ALL
: all relevant fields of the given structure.

For each field specified in *fields*, *t_alloc* will allocate memory for the buffer associated with the field, and initialize the *buf* pointer and *maxlen* field accordingly. The length of the buffer allocated will be based on the same size information that is returned to the user by *t_open(3T)* and *t_getinfo*. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. For any field not specified in *fields*, *buf* will be set to NULL and *maxlen* will be set to zero.

Use of *t_alloc* to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

DIAGNOSTICS

On successful completion, *t_alloc* will return a pointer to the newly allocated structure. It will return NULL on failure. The transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TSYSERR

A system error has occurred during execution of this primitive.

NAME

t_bind – bind an address to a transport endpoint

SYNOPSIS

```
#include <netinet/in.h>
#include <sys/tli.h>
int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

DESCRIPTION

This primitive serves two purposes. First, it associates a protocol address with the transport endpoint specified by *fd*. In doing so, it activates that transport endpoint such that:

1. in connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint; and
2. in connectionless mode, the transport user may send or receive data units through the transport endpoint.

Second, for the connection-mode service only, *t_bind* optionally directs the transport provider to begin accepting connect indications destined for the specified protocol address, where a connect indication specifies that an active user is requesting a connection to the given protocol address.

The *req* and *ret* arguments point to a structure of the following form:

```
struct t_bind {
        struct netbuf addr;
        unsigned      qlen;
```

```
}

struct netbuf {
        unsigned maxlen;
        unsigned len;
        char    *buf;
}
```

where the *addr* field of the *t_bind* structure specifies a protocol address and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

Req is used to request that an address be bound to the given transport endpoint, where the address is represented by the *netbuf* structure. *Len* specifies the number of bytes in the address, *buf* points to a structure of the following form:

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

struct in_addr {
        unsigned long s_addr;
}
```

For internet addresses, *sin_family* must be set to AF_INET, *sin_port* is the port number to bind to, *sin_addr* is the internet address to bind to, and *sin_zero* is currently unused. *maxlen* has no meaning for the *req* argument. The transport provider may bind a different address to the transport endpoint than the address specified in *req*. *Ret* is used to identify the address that

the provider actually bound to the given transport endpoint. For *ret*, *maxlen* specifies the maximum size of the address buffer and *buf* points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address. *Maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the address buffer can hold. If *maxlen* is not large enough to hold the returned address, an error will result.

A user may request a given address to be bound, as specified by the *addr* field of *req*. If the given address is not available, the transport provider may bind the transport endpoint to a different address as returned in *ret*. Also, if no address is specified in *req* (i.e. *req->addr.len* is zero or req is NULL), the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret* (if *ret* is not NULL). The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider TCP. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications.

On return, the *qlen* field in *ret* will contain the negotiated value. If a user attempts to bind the same address to a second transport endpoint, the transport provider will assign another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This will prevent more than one transport endpoint bound to the same protocol address from accepting connection indications.

For connection-mode service, it is possible for a single user process to be both an active and passive transport user, but a user cannot be both active and passive on a single transport endpoint at the same time.

Req may be NULL if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider will assign an address to the transport endpoint. Similarly, *ret* may be NULL if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set both *req* and *ret* to NULL for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

DIAGNOSTICS

t_bind returns 0 on success and -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a

transport endpoint.

TOUTSTATE
: The primitive was issued in the wrong sequence.

TBADADDR
: The specified protocol address was in an incorrect format or contained illegal information.

TNOADDR
: The transport provider could not allocate an address.

TACCES
: The user does not have permission to use the specified address.

TBUFOVFLW
: The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.

TSYSERR
: A system error has occurred during execution of this primitive.

NAME

t_close – close a transport endpoint

SYNOPSIS

```
#include <sys/tli.h>
int t_close(fd)
int fd;
```

DESCRIPTION

The *t_close* primitive informs the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. In addition, *t_close* closes the file associated with the transport endpoint.

t_close should be called from the T_UNBND state. However, this primitive does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, *close*(2) will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

DIAGNOSTICS

t_close returns 0 on success and -1 on failure. The transport error that may occur is:

TBADF

The specified file descriptor does not refer to a transport endpoint.

NAME

t_connect – establish a connection with another transport user

SYNOPSIS

```
#include <sys/tli.h>
int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

DESCRIPTION

This primitive enables a transport user to request a connection to the specified destination transport user. *Fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* are pointers to a structure of the following form:

```
struct t_call {
        struct netbuf addr;
        struct netbuf opt;
        struct netbuf udata;
        int sequence;
}
```

where *sndcall* specifies information needed by the TCP transport provider and *rcvcall* specifies information that is associated with the newly established connection.

In *sndcall*, *addr* is a pointer to sockaddr_in structure which specifies the protocol address of the destination transport user. The *opt*, *udata*, and *sequence* fields are ignored.

In *rcvcall*, *addr* returns the protocol address associated with the responding transport endpoint. The *opt*, *udata*, and *sequence* fields have no meaning for this primitive.

On return, the *addr* field of *rcvcall* will be updated to reflect the address as sent by the destination user in response to the connect request. Thus, the *maxlen* field of *addr* must be set before issuing this primitive to indicate the maximum size of the buffer. However, *rcvcall* may be NULL, in which case no information is given to the user on return from *t_connect*.

By default, *t_connect* executes in synchronous mode and will wait for the destination user's response before returning control to the local user. A successful return (i.e. return value of zero) indicates that the requested connection has been established. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_connect* executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user. The call will return -1 and set *t_errno* to TNODATA to indicate that the connection has not yet been established. In this way, the primitive simply initiates the connection establishment procedure by sending a connect request to the destination transport user. When executed in asynchronous mode, *t_connect* must be used in conjunction with *t_rcvconnect* to confirm that the connection has been established (see *t_rcvconnect(3T)*).

DIAGNOSTICS

t_connect will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE

The primitive was issued in the wrong sequence.

TNODATA
: O_NDELAY was set, so the primitive successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

TBADADDR
: The specified protocol address was in an incorrect format or contained illegal information.

TACCES
: The user does not have permission to use the specified address or options.

TBUFOVFLW
: The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, this error will change the provider's state to T_DATAXFER, and the information to be returned in *rcvcall* will be discarded.

TLOOK
: An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive. The provider's state may change; if so, the state will become T_OUTCON or T_DATAXFER, depending on what point in the connection establishment

procedure the error occurred.

NAME

t_error – produce error message

SYNOPSIS

```
#include <sys/tli.h>
void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

DESCRIPTION

The *t_error* routine produces a message on the standard error output describing the last error encountered during a call to a transport library function. The argument string *errmsg* is a user-supplied error message that gives context to the error. *t_error* will print the user-supplied error message followed by a colon and a standard error message for the current error defined in *t_errno*. *t_errno* is set when errors occur but is not cleared when non-erroneous calls are made.

As an example, if a *t_connect* primitive fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2: Incorrect transport address format
```

where "Incorrect transport address format" identifies the specific error that occurred, and "t_connect failed on fd2" tells the user which primitive failed on which transport endpoint.

To simplify variant formatting of messages, the array of message strings *t_errlist* is provided; *t_errno* can be used as an index in this table to get the message string without the newline. *t_nerr* is the largest message number provided for in the *t_errlist* table.

NAME

t_free – free a library structure

SYNOPSIS

```
#include <sys/tli.h>
int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

DESCRIPTION

The *t_free* routine frees memory previously allocated by *t_alloc*. The routine will free memory for the specified structure and will also free memory for buffers referenced by the structure.

Ptr points to one of the six structure types described for *t_alloc*, and *struct_type* identifies the type of that structure.

t_free will check the *addr*, *opt*, and *udata* fields of the given structure (as appropriate) and free the buffers pointed to by the *buf* field of *netbuf* structure. If *buf* is NULL, *t_free* will not attempt to free memory. After all buffers are freed, *t_free* will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by *t_alloc*.

DIAGNOSTICS

t_free will return 0 on successful completion and -1 on failure. The transport error that may occur is:

TSYSERR

A system error has occurred during execution of this primitive.

NAME

t_getinfo – get protocol-specific service information

SYNOPSIS

```
#include <sys/tli.h>
int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

DESCRIPTION

This primitive returns various characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by *t_open*. This primitive enables a transport user to access this information during any phase of communication. See the description of *t_open(3T)* for the full semantics of this routine.

DIAGNOSTICS

t_getinfo returns 0 on success and -1 on failure. The transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TSYSERR

A system error has occurred during execution of this primitive.

NAME

t_getstate – get the current state

SYNOPSIS

```
#include <sys/tli.h>
int t_getstate(fd)
int fd;
```

DESCRIPTION

The *t_getstate* primitive returns the current state associated with the transport endpoint specified by *fd*. This is the state of the transport provider as seen by the transport user, and can be one of the following:

T_UNBND
: unbound

T_IDLE
: idle

T_OUTCON
: outgoing connection pending

T_INCON
: incoming connection pending

T_DATAXFER
: data transfer

T_OUTREL
: outgoing orderly release (waiting for an orderly release indication)

T_INREL
: incoming orderly release (waiting for an orderly release request)

If the provider is undergoing a state transition when *t_getstate* is called, the routine will fail.

DIAGNOSTICS

t_getstate returns the current state on successful completion and -1 on failure. The transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TSTATECHNG

The transport provider is undergoing a state change.

TSYSERR

A system error has occurred during execution of this primitive.

NAME

t_listen – listen for a connect request

SYNOPSIS

```
#include <sys/tli.h>
int t_listen(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

This primitive listens for a *t_connect(3T)* request from a calling transport user. *Fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* specifies the call structure sent by the calling transport user.

In *call*, *addr* returns the protocol address of the calling transport user. The *opt*, and *udata* fields have no meaning, and *sequence* is a number that uniquely identifies the returned connect indication.

The value of *sequence* may be used by the passive transport user as a means of multi-threading connect indications/responses. It enables the user to listen for multiple connect indications before responding to any of them. When the user eventually responds (see *t_accept(3T)* and *t_snddis(3T)*), the sequence number on the response will indicate to the transport provider which connect indication is being responded to.

Since this primitive returns values for the *addr*, *opt*, and *udata* fields of *call*, the *maxlen* field of each must be set before issuing the *t_listen* to indicate the maximum size of the buffer for each.

By default, *t_listen* executes synchronously and waits for a connect indication to arrive before returning to the user. However, if O_NDELAY is set (via *t_open(3T)*

or *fcntl*), *t_listen* executes asynchronously, reducing to a poll for existing connect indications. If none is available, it fails and returns immediately, freeing the user to do other tasks.

If a user issues *t_listen* in synchronous mode on a transport endpoint that was not bound for listening (i.e. *qlen* was zero on *t_bind(3T)*, the call will wait forever because no connect indications will arrive on that endpoint.

DIAGNOSTICS

t_listen will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
: The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, may change to T_INCON, and the connect indication information to be returned in *call* will be discarded.

TNODATA
: O_NDELAY was set, but no connect indications had been queued.

TLOOK
: An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR

A system error has occurred during execution of this primitive. The provider's state, as seen by the user, may change to T_INCON.

NAME

t_look – look at the current event on a transport endpoint

SYNOPSIS

```
#include <sys/tli.h>
int t_look(fd)
int fd;
```

DESCRIPTION

This primitive returns the current event on the transport endpoint specified by *fd*. The event is the bitwise *or* of the following asynchronous events:

T_LISTEN
: This event occurs when an indication of a connect request from a remote user is received by a transport provider (connection-mode service only).

T_CONNECT
: This event occurs when a connect confirmation is received by a transport provider (connection-mode service only).

T_DATA
: This event occurs when normal data is received by a transport provider.

T_EXDATA
: This event occurs when expedited data is received by a transport provider (connection-mode service only).

T_DISCONNECT
: This event occurs when a disconnect indication is received by a transport provider (connection-mode service only).

T_ORDREL

This event occurs when an orderly release indication is received by a transport provider (connection-mode service with orderly release only).

T_ERROR

This event occurs when a fatal error is generated by the transport provider, thus making the transport endpoint inaccessible.

T_UDERR

This event occurs when an error is found in a previously sent data unit (connectionless-mode service only).

This primitive enables a transport provider to notify a transport user of an asynchronous event when the user is issuing primitives in synchronous mode. Certain asynchronous events require immediate notification of the user, even if the user does not desire asynchronous notification. These special events are indicated by a specific error, TLOOK, on the current or next primitive to be executed. This error tells the user that they should issue a *t_look* to see what event has occurred. They can then respond accordingly. This primitive also enables a transport user to poll a transport endpoint periodically for asynchronous events.

DIAGNOSTICS

t_look returns a value that indicates which of the allowable events has occurred, and returns zero if no event exists. It returns -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TSYSERR
A system error has occurred during execution of this primitive.

NAME

t_open – establish a transport endpoint

SYNOPSIS

```
#include <sys/tli.h>
int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;
```

DESCRIPTION

This primitive establishes a transport endpoint by opening a UNIX file that identifies a particular transport protocol and returns a file descriptor that identifies that endpoint. The following special files are currently recognized:

/dev/tcp TCP (connection-oriented transport layer protocol);
/dev/udp UDP (connection-less transport layer protocol).

Path points to the path name of the file to open, and *oflag* identifies any open flags (as in *open*(2)). Setting *oflag* to O_NDELAY specifies non-blocking operations. *t_open* returns a file descriptor that will be used by all subsequent routines to identify the particular local transport endpoint.

This primitive also returns various characteristics of the underlying transport protocol through *info*. This argument is a pointer to a structure of the following form:

```
struct t_info {
        long addr;
        long options;
```

```
        long tsdu;
        long etsdu;
        long connect;
        long discon;
        long servtype;
}
```

where the fields returned have the following meanings:

addr the maximum size of the transport protocol address;

options the maximum number of bytes of protocol-specific options;

tsdu the maximum size of a transport service data unit (TSDU); TSDU and ETSDU are logical units of data passed between peer users of the transport service, and their size may be limited by the underlying transport provider.

etsdu the maximum size of an expedited transport service data unit (ETSDU);

connect the maximum amount of data allowed on connection establishment primitives;

discon the maximum amount of data allowed on the *t_snddis* and *t_rcvdis* primitives; and *servtype* the service type supported by the transport provider.

For *tsdu* a value of zero indicates that the transport protocol does not support the concept of TSDU although it supports the sending of a data stream with no logical boundaries preserved across a connection. For each field that defines a size in the above structure, a value greater than or equal to zero specifies the limit in bytes of that item, a value of -1 specifies that there is no limit on the size of that item, and a value

of -2 specifies that the transport protocol does not support that feature. The *servtype* field of *info* returns the value T_COTS_ORD for TCP connections and T_CLTS for UDP connections. For both TCP and UDP connections, *t_open* will return 16 (sizeof struct sockaddr_in) for *addr*, 0 for *tsdu* and return -2 for *options*, *connect*, and *discon*. In addition, *etsdu* is set to 1 for TCP connections and -2 for UDP connections.

Info may be set to NULL by the transport user, in which case no protocol information is returned by *t_open*.

t_open must be called as the first step in the initialization of a transport endpoint.

DIAGNOSTICS

t_open returns a valid file descriptor on success and -1 on failure. The transport error that may occur is:

TSYSERR

A system error has occurred during execution of this primitive.

BUGS

A connection is always opened for both reading and writing.

NAME

t_optmgmt – manage options for a transport endpoint

SYNOPSIS

```
#include <sys/socket.h>
#include <sys/tli.h>
int t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

DESCRIPTION

The *t_optmgmt* primitive enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. *Fd* identifies a bound transport endpoint.

The *req* and *ret* arguments point to a structure of the following form:

```
struct t_optmgmt {
        struct netbuf opt;
        long    flags;
}
```

where the *opt* field identifies protocol options and the *flags* field is used to specify the action to take with those options.

Req is used to request a specific action of the provider and to send options to the provider. *Len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider will return flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options

returned. *Maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold.

The *flags* field of *req* can specify one of the following actions:

T_NEGOTIATE
: This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*.

T_CHECK
: This action enables the user to verify whether the options specified in *req* are supported by the transport provider. On return, the *flags* field of *ret* will have either T_SUCCESS or T_FAILURE set to indicate to the user whether the options are supported. These flags are only meaningful for the T_CHECK request.

T_DEFAULT
: This action enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*.

The options are represented by a *netbuf* structure where *opt.buf* is a pointer to a structure of the form

```
struct {
        unsigned short name;
        unsigned short val;
}
```

where *name* could be any one of the following:

SO_DEBUG
Turn on debugging information recording;

SO_REUSEADDR
Allow local address reuse;

SO_KEEPALIVE
Keep connections alive;

SO_DONTROUTE
Just use interface addresses;

SO_USELOOPBACK
Bypass hardware when possible;

SO_LINGER
Linger on close if data present;

SO_DONTLINGER
Don't linger on close.

val must be supplied if SO_LINGER option is specified. For all other options, *val* is meaningless.

DIAGNOSTICS

t_optmgmt returns 0 on success and -1 on failure. Transport errors that may occur are:

TBADF
The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE
The primitive was issued in the wrong sequence.

TACCES
The user does not have permission to negotiate the specified options.

TBADOPT
The specified protocol options were in an

incorrect format or contained illegal information.

TBADFLAG
: An invalid flag was specified.

TBUFOVFLW
: The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.

TSYSERR
: A system error has occurred during execution of this primitive.

NAME

t_rcv – receive data or expedited data sent over a connection

SYNOPSIS

```
#include <sys/tli.h>
int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

DESCRIPTION

This primitive receives either normal or expedited data. *Fd* identifies the local transport endpoint through which data will arrive, *buf* points to a receive buffer where user data will be stored, *nbytes* specifies the size of the receive buffer, and *flags* specifies optional flags as described below.

By default, *t_rcv* operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NDELAY is set (via *t_open(3T)* or *fcntl*), *t_rcv* will execute in asynchronous mode and will not wait if no data is available, but will fail.

t_rcv returns available data. If the data returned is expedited, T_EXPEDITED will be set in *flags* on return from this routine. The TCP transport provider supports the transfer of a maximum of one byte of expedited data.

On successful completion, *t_rcv* returns the number of bytes received, and it returns -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a

transport endpoint.

TNODATA
: O_NDELAY was set, but no data is currently available from the transport provider.

TLOOK
: An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive.

NAME

t_rcvconnect – receive the confirmation from a connect request

SYNOPSIS

```
#include <sys/tli.h>
int t_rcvconnect(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

This primitive enables a calling transport user to determine the status of a previously sent connect request. It is used in conjunction with *t_connect* to establish a connection in asynchronous mode. The connection will be established on successful completion of this primitive.

Fd identifies the local transport endpoint where communication is to be established, and *call* specifies the call structure returned by the destination transport user.

In *call*, *addr* returns the protocol address associated with the responding transport endpoint. The *opt*, *udata* and *sequence* fields have no meaning for this primitive.

By default, *t_rcvconnect* executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr*, *opt*, and *udata* fields will reflect values as sent by the responding transport user. Thus, the *maxlen* field of each argument must be set before issuing this primitive to indicate the maximum size of the buffer for each. However, *call* may be NULL, in which case no information is given to the user on return from *t_rcvconnect*.

If O_NDELAY is set (via *t_open(3T)* or *fcntl*), *t_rcvconnect* executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If

none are available, it fails and returns immediately without waiting for the connection to be established; *t_rcvconnect* must then be re-issued at a later time to complete the connection establishment phase and retrieve the negotiated values of these parameters.

DIAGNOSTICS

t_rcvconnect will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
: The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. This error will change the state of the provider as seen by the user to T_DATAXFER, and the connect information to be returned in *call* will be discarded.

TNODATA
: O_NDELAY was set, but a connect confirmation has not yet arrived.

TLOOK
: An asynchronous event has occurred on this transport connection and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive. The provider's state, as seen by the user, may change to T_DATAXFER.

NAME

t_rcvdis – retrieve information from disconnect

SYNOPSIS

```
#include <sys/tli.h>
t_rcvdis(fd, discon)
int fd;
struct t_discon *discon;
```

DESCRIPTION

This primitive is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. *Fd* identifies the local transport endpoint where the connection existed, and *discon* points to a structure of the following form:

```
struct t_discon {
        struct netbuf udata;
        int     reason;
        int     sequence;
}
```

where *reason* specifies the reason for the disconnect through a protocol-dependent reason code, *udata* field is ignored, and *sequence* may identify an outstanding connect indication with which the disconnect is associated.

The *sequence* number is only meaningful when *t_rcvdis* is issued by a passive transport user who has executed one or more *t_listen(3T)* primitives and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

t_rcvdis should only be issued after the user is notified of a disconnect indication.

Discon may be NULL if *t_rcvdis* is not issued during the connection establishment phase. If *discon* is NULL, any user data associated with the disconnect will be discarded. If a user receives a disconnect indication and has retrieved more than one outstanding connect indication (via *t_listen(3T)*, *discon* should not be NULL; if *discon* is NULL in this case, the user will be unable to identify with which connect indication the disconnect is associated.

The possible values for the *reason* field are :

ECONNABORTED
: Connection aborted.

EHOSTDOWN
: Destination host is down.

EHOSTUNREACH
: Destination host is unreachable.

ENETUNREACH
: The network is unreachable.

ECONNREFUSED
: The destination host refused the connection request.

ECONNRESET
: The connection was reset by the destination host.

ETIMEDOUT
: The connection timed out.

DIAGNOSTICS

t_rcvdis will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TNODIS
: No disconnect indication currently exists on the specified transport endpoint.

TBUFOVFLW
: The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, will change to T_IDLE , and the disconnect indication information to be returned in *discon* will be discarded.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive. The provider's state, as seen by the user, may change to T_IDLE .

NAME

t_rcvrel – acknowledge receipt of an orderly release indication

SYNOPSIS

```
#include <sys/tli.h>
t_rcvrel(fd)
int fd;
```

DESCRIPTION

This primitive is used to inform the transport provider that the transport user has received an orderly release indication. *Fd* identifies the local transport endpoint where the connection exists.

t_rcvrel should only be issued if the user is notified of an orderly release indication.

After receiving an orderly release indication, a user may not attempt to receive more data. However, if the user has not yet issued a *t_sndrel(3T)*, they can continue to send data over the connection. The connection will remain until both users have issued a *t_sndrel(3T)* and have received an orderly release indication.

The UDP transport provider does not support this service.

DIAGNOSTICS

t_rcvrel will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TNOREL

No orderly release indication currently exists on the specified transport endpoint.

TLOOK
: An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive. The provider's state may change; if so, the state will become T_INREL or T_IDLE, depending on the state the primitive was issued from.

NAME

t_rcvudata – receive a data unit

SYNOPSIS

```
#include <sys/tli.h>
int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

DESCRIPTION

This primitive is used to receive a data unit from another transport user in connectionless mode. *Fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, and *flags* is ignored.

In *unitdata*, on return from this call, *addr* specifies the protocol address of the sending user, and *udata* specifies the user data that was received.

The *maxlen* field of *addr* and *opt* must be set before issuing this primitive to indicate the maximum size of the buffer for each.

By default, *t_rcvudata* operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if O_NDELAY is set (via *t_open(3T)* or *fcntl*), *t_rcvudata* will execute in asynchronous mode and will not wait if no data units are available, but will fail.

DIAGNOSTICS

t_rcvudata returns 0 on successful completion and -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TNODATA
: O_NDELAY was set, but no data units are currently available from the transport provider.

TBUFOVFLW
: The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded.

TLOOK
: An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive.

NAME

t_rcvuderr – receive a unit data error indication

SYNOPSIS

```
#include <sys/tli.h>
int t_rcvuderr(fd, uderr)
int fd;
struct t_uderr *uderr;
```

DESCRIPTION

This primitive is used to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. *Fd* identifies the local transport endpoint through which the error report will be received, and *uderr* points to a structure of the following form:

```
struct t_uderr {
        struct netbuf addr;
        struct netbuf opt;
        long    error;
}
```

which identifies the error.

In *uderr*, on return from this call, *addr* specifies the destination protocol address of the erroneous data unit, *opt* identifies protocol-specific options that were associated with the data unit, and *error* specifies a protocol-dependent error code.

The *maxlen* field of *addr* and *opt* must be set before issuing this primitive to indicate the maximum size of the buffer for each.

t_rcvuderr should only be issued after the user is notified of an erroneous data unit.

Uderr may be set to NULL by the user if the user does not care to identify the data unit that produced an error. In this case, *t_rcvuderr* will simply clear the error indication without reporting any information to the user.

DIAGNOSTICS

t_rcvuderr returns 0 on successful completion and -1 on failure. Transport errors that may occur are:

TBADF
The specified file descriptor does not refer to a transport endpoint.

TNOUDERR
No unit data error indication currently exists on the specified transport endpoint.

TBUFOVFLW
The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.

TNOTSUPPORT
This primitive is not supported by the underlying transport provider.

TSYSERR
A system error has occurred during execution of this primitive.

NAME

t_snd – send data or expedited data over a connection

SYNOPSIS

```
#include <sys/tli.h>
int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

DESCRIPTION

This primitive is used to send either normal or expedited data during the data transfer phase. *Fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below.

By default, *t_snd* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY is set (via *t_open(3T)* or *fcntl*), *t_snd* will execute in asynchronous mode, and will not wait under such conditions, but will fail.

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if O_NDELAY is set, it is possible that only part of the data will actually be accepted by the transport provider. This will be indicated by a return value that is less than the value of *nbytes*.

If *nbytes* is zero, no data will be passed to the transport provider; *t_snd* will return zero.

If T_EXPEDITED is set in *flags*, the data will be sent as expedited data, and will be subject to the interpretations of the transport provider.

If T_MORE is set in *flags*, it is ignored.

If *t_snd* is issued from the T_IDLE state, the provider will silently discard the data. If *t_snd* is issued from any state other than T_DATAXFER or T_IDLE, the provider will generate an EPROTO protocol error.

t_snd does not check for a broken connection before passing data to the provider. If a disconnect occurs (thereby changing the current state to T_IDLE), the provider will discard the data without notifying the user. Therefore, the transport user must monitor the incoming side of the connection for a disconnect event. Similarly, the user may wish to monitor the connection for an incoming orderly release event.

DIAGNOSTICS

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider, and it returns -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TFLOW
: O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR

A system error has occurred during execution of this primitive.

NAME

t_snddis – send user-initiated disconnect request

SYNOPSIS

```
#include <sys/tli.h>
int t_snddis(fd, call)
int fd;
struct t_call *call;
```

DESCRIPTION

This primitive is used to request the release of a connection without negotiating that release. In other words, it initiates an abortive release. It may also be used to reject a connect request during the connection establishment phase. *Fd* identifies the local transport endpoint where the connection exists, and *call* specifies the call structure to be sent to the remote user.

Call may be the structure returned by *t_listen(3T)* when rejecting a connect request, or it may be the structure returned by *t_connect(3T)*, *t_rcvconnect(3T)*, or *t_listen(3T)* when aborting a connection from the data transfer phase. The *addr* and *opt* fields of *call* retain the values returned by *t_connect(3T)*, *t_rcvconnect(3T)*, or *t_listen(3T)* *sequence* is used when rejecting a connect indication to uniquely identify the indication to the transport provider.

t_snddis will generate a disconnect indication to the user process on the other side of the connection. This indication must be handled by the user on the other side of the connection.

When rejecting a connect indication during the connection establishment phase, the value of *sequence* may be used by the passive transport user as a means of multithreading connect indications/responses. It enables the user to listen for multiple connect indications (via

t_listen(3T) before responding to any of them. When the user eventually responds, *sequence* tells the transport provider with which connect indication the response is associated. The value of *sequence* will be ignored if *t_snddis* is not being used to reject a connection (i.e. there are no outstanding connect indications).

Call may be NULL if *t_snddis* is issued after a connection has been established (i.e. *call* is not being used to reject a connection), in which case no user data is sent to the remote user. *t_snddis* will fail if *call* is NULL and the current state is not T_DATAXFER, because invalid call information will be sent to the provider.

DIAGNOSTICS

t_snddis will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE

The primitive was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost.

TBADSEQ

An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost.

TLOOK

An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive. The provider's state may change to T_IDLE.

NAME

t_sndrel – initiate an orderly release (optional)

SYNOPSIS

```
#include <sys/tli.h>
int t_sndrel(fd)
int fd;
```

DESCRIPTION

This primitive is used to indicate to the transport provider that the transport user has no more data to send. *Fd* identifies the local transport endpoint where the connection exists.

This primitive is used in the orderly release of a connection. *t_sndrel* will generate an orderly release indication to the user process on the other side of the connection. This indication must be acknowledged to the transport provider by the user on the other side of the connection (see *t_rcvrel*).

After issuing *t_sndrel(3T)*, the user may not send any more data over the transport connection. However, if the user has not received an orderly release indication, they should continue receiving incoming data until such an indication arrives. The connection will remain until both users have issued a *t_sndrel* and have received notification that the other user has requested the orderly release of the connection.

The UDP transport provicer does not support this service.

If *t_sndrel* is issued from an invalid state, the provider will generate an EPROTO protocol error.

t_sndrel does not check for a broken connection before passing data to the provider. Therefore, the transport user must monitor the incoming side of the connection

for a disconnect event, and should not issue *t_sndrel* if this event has occurred.

DIAGNOSTICS

t_sndrel will return 0 on success and -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TFLOW
: O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting the primitive at this time.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive.

NAME

t_sndudata – send a data unit

SYNOPSIS

```
#include <sys/tli.h>
int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

DESCRIPTION

This primitive is used to send a data unit to another transport user in connectionless mode. *Fd* identifies the local transport endpoint through which data will be sent, and *unitdata* points to a structure of the following form:

```
struct t_unitdata {
        struct netbuf addr;
        struct netbuf opt;
        struct netbuf udata;
}
```

which provides all information needed by the transport provider to send the data.

In *unitdata*, *addr* specifies the protocol address of the destination user, and *opt* and *udata* fields are ignored.

By default, *t_sndudata* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_sndudata* will execute in asynchronous mode, and will not wait under such conditions, but will fail.

This primitive is not acknowledged by the transport provider. Any errors associated with the request (e.g. bad address format), will be reported through a unit

data error indication (see *t_rcvudern(3T)*). However, if *t_sndudata* is issued from an invalid state, the provider will generate an EPROTO error.

If the *len* field of *udata* is zero, no data unit will be passed to the transport provider; *t_sndudata* will not send zero-length data units.

t_sndudata does not check for a unit data error indication before transmitting the given data unit. Therefore, a user who is sending data units but not receiving data units is responsible for monitoring the transport endpoint for unit data error indications. Otherwise, the user may never be notified of errors on previously sent data units.

DIAGNOSTICS

t_sndudata will return 0 on successful completion and -1 on failure. Transport errors that may occur are:

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TFLOW
: O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time.

TNOTSUPPORT
: This primitive is not supported by the underlying transport provider.

TSYSERR
: A system error has occurred during execution of this primitive.

NAME

t_sync – synchronize transport library

SYNOPSIS

```
#include <sys/tli.h>
int t_sync(fd)
int fd;
```

DESCRIPTION

The *t_sync* routine synchronizes the state of the transport library interface with that of the underlying transport provider for the transport endpoint specified by *fd*. This routine is useful when two or more processes are communicating over the same transport endpoint and have to update their own process data space to match the current state of the provider.

If, for example, a process *forks* a new process and issues an *exec*, the new process must issue a *t_sync* to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the provider's state. Furthermore, if one of the processes then issues a primitive that causes a provider state change, the other process must issue *t_sync* to update the library's notion of the provider's state.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they must coordinate their activities so as not to violate the state of the provider. *t_sync* returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only guaranteed among cooperating processes; it is possible that a process could change the provider's state after a *t_sync* is issued. The states returned are the same constants returned by

t_getstate(3T).

If the provider is undergoing a state transition when *t_sync* is called, the routine will fail.

DIAGNOSTICS

On successful completion, *t_sync* returns the state of the transport provider, and it returns -1 on failure. Transport errors that may occur are:

TBADF
The specified file descriptor does not refer to a transport endpoint.

TSTATECHNG
The transport provider is undergoing a state change.

TSYSERR
A system error has occurred during execution of this primitive.

NAME

t_unbind – disable a transport endpoint

SYNOPSIS

```
#include <sys/tli.h>
int t_unbind(fd)
int fd;
```

DESCRIPTION

The *t_unbind* primitive disables the transport endpoint specified by *fd*. On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

DIAGNOSTICS

t_unbind returns 0 on success and -1 on failure. Transport errors that may occur are:

TBADF

The specified file descriptor does not refer to a transport endpoint.

TOUTSTATE

The primitive was issued in the wrong sequence.

TLOOK

An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TSYSERR

A system error has occurred during execution of this primitive. The provider's state may change to T_UNBND

NAME

intro – introduction to Wollongong Socket Compatibility Library

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes the functions that make up the Wollongong Socket Compatibility Library. These functions constitute a self-contained interface to the transport level protocols, *tcp(4)* and *udp(4)*.

They may be used in conjunction with the functions in Section (3N) but not with those in Section (3T). They are fully compatible with the functions of the same names in the 4.2 BSD release of the UNIX operating system. Most of the programs in Sections (1) and (1M) are based on the calls in this section.

A *socket* is an endpoint for communication between processes. Each *socket* has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a *socket* require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of *socket* types; consult *socket(3W)* for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each *socket* has an address chosen from the

address family in which the *socket* was created.

Certain semantics of the basic *socket* abstractions are protocol specific. All protocols are expected to support the basic model for their particular *socket* type, but may, in addition, 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.

The Internet family, *inet(4)*, provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types.

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 type SOCK_RAW. The ICMP message protocol is not directly accessible.

Sockets utilizing 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(3W)* system call must be used after binding the socket with the *bind(3W)* system call. Only passive sockets may use the *accept(3W)* call to accept incoming connections. Only active sockets may use the *connect(3W)* 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. Normally this address corresponds to the peer entity's network.

UDP is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the *sendto(3w)* and *recvfrom(3w)* calls, though the *connect(3W)* call may also be used to fix the destination for future packets (in which case *recv(3W)* and *send(3W)* or the *read(2)* or *write(2)* system calls may be used).

IP may be accessed through a "raw socket" when developing new protocols, or special purpose applications. IP sockets are connectionless, and are normally used with *sendto(3W)* and *recvfrom(3W)*, though the *connect(3W)* call may also be used to fix the destination for future packets (in which case *recv(3W)* and *send(3W)* or *read(2)* and *write(2)* system calls may be used).

DIAGNOSTICS

Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always "– 1"; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable *errno*, which is not cleared on successful calls. Thus *errno* should be tested only after an

error has occurred.

The following is a complete list of the errors and their names as given in *<errno.h>*.

1-34 Standard

35 EWOULDBLOCK Operation would block
An operation which would cause a process to block was attempted on a object in non-blocking mode (see *ioctl* (2)).

36 EINPROGRESS Operation now in progress
An operation which takes a long time to complete (such as a *connect(3W)*) was attempted on a non-blocking object (see *ioctl*(2)).

37 EALREADY Operation already in progress
An operation was attempted on a non-blocking object which already had an operation in progress.

38 ENOTSOCK Socket operation on non-socket
Self-explanatory.

39 EDESTADDRREQ Destination address required
A required address was omitted from an operation on a socket.

40 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer.

41 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 ENOPROTOOPT Bad protocol option
A bad option was specified in a *getsockopt*(3W) or *setsockopt*(3W) call.

43 EPROTONOSUPPORT Protocol not supported
The protocol has not been configured into the system or no implementation for it exists.

44 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

45 EOPNOTSUPP Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

46 EPFNOSUPPORT Protocol family not supported
The protocol family has not been configured into the system or no implementation for it exists.

47 EAFNOSUPPORT Address family not supported by protocol family
An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

48 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

49 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

50 ENETDOWN Network is down
A socket operation encountered a dead network.

51 ENETUNREACH Network is unreachable
A socket operation was attempted to an unreachable network.

52 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

53 ECONNABORTED Software caused connection abort
A connection abort was caused internal to your host machine.

54 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from the peer executing a *shutdown*(3W) call.

55 ENOBUFS No buffer space available
An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.

56 EISCONN Socket is already connected
A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination other than the connected party.

57 ENOTCONN Socket is not connected
An request to send or receive data was disallowed because the socket is not connected.

58 ESHUTDOWN Can't send after socket shutdown
A request to send data was disallowed because the socket had already been shut down with a previous *shutdown(3W)* call.

59 *unused*

60 ETIMEDOUT Connection timed out
A *connect* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

61 ECONNREFUSED Connection refused
No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service which is inactive on the foreign host.

DEFINITIONS

Descriptor
An integer assigned by the system when a file is referenced by *open*(2), *dup*(2), or *pipe*(2) or a socket is referenced by *socket(3W)* or *socketpair(3W)* which uniquely identifies an access path to that file or socket from a given process or any of its children.

SEE ALSO

intro(3)
open(2), *dup(2)*, *pipe(2)*, *perror(3)*, UNIX System V Programmer Reference Manual

NAME

accept – accept a connection on a socket

SYNOPSIS

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

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument *s* is a socket which has been created with *socket(3W)*, bound to an address with *bind(3W)*, and is listening for connections after a *listen(3W)*. *Accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter which is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based

socket types, currently with SOCK_STREAM.

It is possible to *select(3W)* a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

The call returns "– 1" on error. If it succeeds it returns a non-negative integer which is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(3W), connect(3W), listen(3W), select(3W), socket(3W)

NAME

bind – bind a name to a socket

SYNOPSIS

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

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

Bind assigns a name to an unnamed socket. When a socket is created with *socket(3W)* it exists in a name space (address family) but has no name assigned. *Bind* requests the name *name*, be assigned to the socket.

NOTES

Binding a name in the UNIX domain creates a socket in the file system which must be deleted by the caller when it is no longer needed (using *unlink*(2)). The file created is a side-effect of the current implementation, and will not be created in future versions of the UNIX domain.

The rules used in name binding vary between communication domains. Consult the manual entries in Section 4 for detailed information.

RETURN VALUE

If the bind is successful, a "0" value is returned. A return value of "– 1" indicates an error, which is further specified in the global *errno*.

ERRORS

The *bind* call will fail if:

[EBADF] *S* is not a valid descriptor.

[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

SEE ALSO

connect(3W), listen(3W), socket(3W), getsockname(3W) unlink(2), **UNIX System V Programmer Reference Manual**

NAME

connect – initiate a connection on a socket

SYNOPSIS

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

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name* which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way.

RETURN VALUE

If the connection or binding succeeds, then "0" is returned. Otherwise a "– 1" is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified

	address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EWOULDBLOCK]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select(3W)* the socket while it is connecting by selecting it for writing.

SEE ALSO

accept(3W), select(3W), socket(3W), getsockname(3W)

NAME

gethostid, sethostid – get/set unique identifier of current host

SYNOPSIS

```
hostid = gethostid()
int hostid;

sethostid(hostid)
int hostid;
```

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor which is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the superuser and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(3W)

BUGS

32 bits for the identifier is too small.

NAME

gethostname, sethostname – get/set name of current host

SYNOPSIS

```
gethostname(name, namelen)
char *name;
int namelen;

sethostname(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

RETURN VALUE

If the call succeeds a value of "0" is returned. If the call fails, then a value of "– 1" is returned and an error code is placed in the global variable *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller was not the superuser.

SEE ALSO

hostname(1), gethostid(3W)

BUGS

Host names are limited to 31 characters.

NAME

getpeername – get name of connected peer

SYNOPSIS

```
getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A "0" is returned if the call succeeds, "– 1" if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(3W), socket(3W), getsockname(3W)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getpeername* returns a zero length name.

NAME

getsockname – get socket name

SYNOPSIS

```
getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

Getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A "0" is returned if the call succeeds, "– 1" if it fails.

ERRORS

The call succeeds unless:

[EBADF] The argument *s* is not a valid descriptor.

[ENOTSOCK] The argument *s* is a file, not a socket.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[EFAULT] The *name* parameter points to memory not in a valid part of the process address space.

[EADDRNOTAVAIL] Socket not bound.

SEE ALSO

bind(3W), socket(3W)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getsockname* returns a zero length name.

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

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

getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return

to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as "0".

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. Options at other protocol levels vary in format and name, consult the appropriate entries in Section 4.

RETURN VALUE

Getsockopt returns "0" if the call succeeds and the specified option is set, otherwise "– 1". *Setsockopt* returns "0" if the call succeeds, "– 1" if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown and thus has not been set.
[EFAULT]	The options are not in a valid part of the process address space.
[ENOBUFS]	No buffer space is available.
[EINVAL]	Invalid option or level specified.

BUGS

Currently SOL_SOCKET is the only *level* that is implemented.

Optval and *optlen* are meaningful only for SO_LINGER option and are required for *setsockopt*.

SEE ALSO

socket(3W), getprotoent(3N)

NAME

listen – listen for connections on a socket

SYNOPSIS

listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket(3W)*, a backlog for incoming connections is specified with *listen(3W)* and then the connections are accepted with *accept(3W)*. The *listen* call applies only to sockets of type SOCK_STREAM.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client will receive an error with an indication of ECONNREFUSED.

RETURN VALUE

A "0" return value indicates success; "– 1" indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

accept(3W), connect(3W), socket(3W)

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

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

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

cc = recvmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Recv, *recvfrom*, and *recvmsg* are used to receive messages from a socket.

The *recv* call may be used only on a *connected* socket (see *connect(3W)*), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether or not it is in a connected state.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is

returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; see *socket(3W)*.

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl*(2)) in which case a *cc* of "–1" is returned with the external variable errno set to EWOULDBLOCK.

The *select(3W)* call may be used to determine when more data arrives.

The *flags* argument to a *recv(3W)* call is formed by *or*'ing one or more of the values,

```
#defineMSG_PEEK  0x1 /* peek at incoming message */
#defineMSG_OOB   0x2 /* process out-of-band data */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in *<sys/socket.h>*:

```
struct msghdr {
caddr_t msg_name;       /* optional address */
int     msg_namelen;    /* size of address */
struct  iov *msg_iov;   /* scatter/gather array */
int     msg_iovlen;     /* # elements in msg_iov */
caddr_t msg_accrights;  /* access rights sent/received */
int     msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *read(3W)*.

Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*.

RETURN VALUE

These calls return the number of bytes received, or "-1" if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.

SEE ALSO

read(2), send(3W), socket(3W) ioctl(2), UNIX System V Programmer Reference Manual

NAME

select – synchronous i/o multiplexing

SYNOPSIS

```
#include <sys/time.h>

nfound = select(nfds, readfds, writefds, execptfds,
        timeout)
int nfound, nfds, *readfds, *writefds, *execptfds;
struct timeval *timeout;
```

DESCRIPTION

Select examines the i/o descriptors specified by the bit masks *readfds*, *writefds*, and *execptfds* to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. File descriptor *f* is represented by the bit "1<<f" in the mask. *Nfds* desciptors are checked, i.e. the bits from 0 through *nfds*-1 in the masks are examined. *Select* returns, in place, a mask of those descriptors which are ready. The total number of ready descriptors is returned in *nfound*.

Timeout uses a *timeval* structure to specify a maximum interval to wait for the selection to complete. This structure has the following form, as defined in <*sys/time.h*>:

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

If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero valued *timeval* structure.

Any of *readfds*, *writefds*, and *execptfds* may be given as "0" if no descriptors are of interest.

RETURN VALUE

Select returns the number of descriptors which are contained in the bit masks, or "-1" if an error occurred. If the time limit expires then *select* returns "0".

ERRORS

An error return from *select* indicates:

[EBADF] One of the bit masks specified an invalid descriptor.

[EINTR] A signal was delivered before any of the selected for events occurred or the time limit expired.

SEE ALSO

accept(3W), connect(3W), recv(3W), send(3W)
read(2) and *write(2)*, UNIX System V Programmer Reference Manual.

BUGS

The descriptor masks are always modified on return, even if the call returns as the result of the timeout.

NAME

send, sendto, sendmsg – send a message from a socket

SYNOPSIS

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

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

Send, *sendto*, and *sendmsg* are used to transmit a message to another socket. *Send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of "– 1" indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking i/o mode. The *select(3W)* call may be used to determine when it is possible to send more data.

The *flags* parameter may be set to MSG_OOB to send "out-of-band" data on sockets which support this notion (e.g. SOCK_STREAM).

See *recv(3W)* for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or "– 1" if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.

SEE ALSO

recv(3W), socket(3W)

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

shutdown(s, how)
int s, how;

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is "0", then further receives will be disallowed. If *how* is "1", then further sends will be disallowed. If *how* is "2", then further sends and receives will be disallowed.

DIAGNOSTICS

A "0" is returned if the call succeeds, "– 1" if it fails.

ERRORS

The call succeeds unless:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a file, not a socket.
[EINVAL]	Invalid value specified for *how*.

SEE ALSO

connect(3W), socket(3W)

NAME

socket – create an endpoint for communication

SYNOPSIS

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

s = socket(af, type, protocol)
int s, af, type, protocol;
```

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies an address format with which addresses specified in later operations using the socket should be interpreted. These formats are defined in the include file <*sys/socket.h*>. The currently understood format is

AF_UNIX	(UNIX path names),
AF_INET	(ARPA Internet addresses),
AF_PUP	(Xeros PUP-I Internet addresses), and
AF_IMPLINK	(IMP "host at IMP" addresses).

The socket has the indicated *type* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams with an out-of-band data transmission mechanism. A SOCK_DGRAM socket supports datagrams

(connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network interfaces. The types SOCK_RAW, which is available only to the superuser, and SOCK_SEQPACKET and SOCK_RDM, which are planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type using a given address format. However, it is possible that many protocols may exist in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *services(5)* and *protocols(5)*.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect(3W)* call. Once connected, data may be transferred using *read(2)* and *write(2)* calls or some variant of the *send(3W)* and *recv(3W)* calls. When a session has been completed a *close(3W)* may be performed. Out-of-band data may also be transmitted as described in *send(3W)* and received as described in *recv(3W)*.

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an

error with "-1" returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. five minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send(3W)* calls. It is also possible to receive datagrams at such a socket with *recv(3W)*.

An *ioctl(2)* call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>* and explained below. *Setsockopt* and *getsockopt(3W)* are used to set and get options, respectively.

SO_REUSEADDR	allow local address reuse
SO_KEEPALIVE	keep connections alive
SO_DONTROUTE	do no apply routing on outgoing messages
SO_LINGER	linger on close if data present
SO_DONTLINGER	do not linger on close

SO_REUSEADDR indicates the rules used in validating addresses supplied in a *bind(3W)* call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to

these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address. SO_LINGER and SO_DONTLINGER control the actions taken when unsent messags are queued on socket and a *close(3W)* is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_DONTLINGER is specified and a *close* is issued, the system will process the close in a manner which allows the process to continue as quickly as possible.

RETURN VALUE

A "-1" is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EAFNOSUPPORT] The specified address family is not supported in this version of the system.

[ESOCKTNOSUPPORT] The specified socket type is not supported in this address family.

[EPROTONOSUPPORT] The specified protocol is not supported.

[EMFILE]	The per-process descriptor table is full.
[ENOBUFS]	No buffer space is available. The socket cannot be created.

SEE ALSO

accept(3W), bind(3W), connect(3W), getsockname(3W), getsockopt(3W), listen(3W), recv(3W), select(3W), send(3W), shutdown(3W), socketpair(3W)
ioctl(2), **UNIX System V Programmer Reference Manual**

NAME

networking – introduction to networking facilities

SYNOPSIS

```
#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>
```

DESCRIPTION

This section describes the networking facilities available in 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, though the current protocol implementations do not. A protocol-family is normally comprised of a number of protocols. 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.

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 only the DARPA Internet protocols fully. Raw socket interfaces are provided to IP protocol layer of the DARPA Internet.

ADDRESSING

Associated with each protocol-family is an *address format*. The following address format is used by the system:

```
#define AF_INET  2 /* internetwork: UDP, TCP, etc. */
```

ROUTING

The network facilities provide limited packet routing. A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two *ioctl(2)* commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by superuser.

A routing table entry has the following form, as defined in *<net/route.h>*;

```
struct rtentry {
        u_long      rt_hash;
        struct      sockaddr rt_dst;
        struct      sockaddr rt_gateway;
        short       rt_flags;
        short       rt_refcnt;
        u_long      rt_use;
        struct      ifnet *rt_ifp;
};
```

with *rt_flags* defined from,

```
#define RTF_UP    0x1    /* route usable */
```

```
#define RTF_GATEWAY 0x2 /* destination is a gateway */
#define RTF_HOST 0x4 /* host entry (net otherwise) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted, each network interface autoconfigured installs a routing table entry when it wishes to have packets sent through it. Normally the interface specifies the route through a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface may be requested to address the packet to an entity different from the eventual recipient (i.e. the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the resources associated with it will not be reclaimed until further references to it are released.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existant entry, or ENOBUFS if insufficient resources were available to install a new route.

User processes read the routing tables through the */dev/kmem* device.

The *rt_use* field contains the number of packets sent along the route. This value is used to select among multiple routes to the same destination. When multiple routes to the same destination exist, the least used route is selected.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

INTERFACES

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, *lo(4)*, do not.

At load time each interface which has underlying hardware support makes itself known to the system. Once the interface has acquired its address it is expected to install a routing table entry so that messages may be routed through it. Most interfaces require some part of their address specified with an SIOCSIFADDR *ioctl* before they will allow traffic to flow through them. On interfaces where the network-link layer address mapping is static, only the network number is taken from the *ioctl*; the remainder is found in a hardware specific manner. On interfaces which provide dynamic network-link layer address mapping facilities (e.g. 10Mb/s Ethernets), the entire address specified in the *ioctl* is used.

The following *ioctl* calls may be used to manipulate network interfaces. Unless specified otherwise, the request takes an *ifrequest* structure as its parameter. This structure has the form

```
struct  ifreq {
        char    ifr_name[16];  /* name of interface
        union {                                (e.g. "lo0") */
```

```
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                short   ifru_flags;
        } ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr /* other end of p-to-p link */
#define ifr_flags ifr_ifru.ifru_flags /* flags */
};
```

SIOCSIFADDR
: Set interface address. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
: Get interface address.

SIOCSIFDSTADDR
: Set point to point address for interface.

SIOCGIFDSTADDR
: Get point to point address for interface.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS
: Get interface flags.

SIOCGIFCONF
: Get interface configuration list. This request takes an *ifconf* structure (see below) as a value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct  ifconf {
        int     ifc_len;  /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf  ifc_ifcu.ifcu_buf   /* buffer address */
#define ifc_req  ifc_ifcu.ifcu_req   /* array of structures returned */
};
```

SEE ALSO

socket(3W), *routed(1M)*

ioctl(2), **UNIX System V Programmer Reference Manual**

config(1M), **UNIX System V Administrator Reference Manual**

NAME

arp – Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses on a local area network. It is used by the 10Mb/s Ethernet interface driver and is not directly accessible to users.

ARP caches Internet-Ethernet address mappings. When the 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 messages are transmitted. ARP itself is not Internet or Ethernet specific; this implementation, however, is. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

ARP watches passively for hosts impersonating the local host (i.e. a host which responds to an ARP mapping request for the local host's address) and will, optionally, periodically probe a network looking for impostors.

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x %x %x %x %x %x . ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

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 atop the Internet Protocol (IP) network layer, and utilizing the Internet address format.

ADDRESSING

Internet addresses are four byte quantities, stored in network standard format. The include file *<netinet/in.h>* defines an Internet address as a discriminated union.

PROTOCOLS

The Internet protocol family is comprised of the Internet Protocol itself, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP).

SEE ALSO

tcp(4), udp(4), ip(4)

CAVEAT

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

NAME

ip – Internet Protocol

SYNOPSIS

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

s = socket(AF_INET, SOCK_RAW, 0);
```

DESCRIPTION

IP is the network layer protocol used by the Internet protocol family.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with). Likewise, incoming packets have their IP header stripped before being sent to the user.

SEE ALSO

send(3W), recv(3W), intro(4), inet(4)

BUGS

One should be able to send and receive IP options.

The protocol should be settable after socket creation.

NAME

lo – software loopback network interface

SYNOPSIS

pseudo-device loop

DESCRIPTION

The *loop* interface is a software loopback mechanism which may be used for performance analysis, software testing, and/or local communication. By default, the loopback interface is accessible at address 127.0.0.1 (non-standard); this address may be changed with the SIOCSIFADDR *ioctl*.

DIAGNOSTICS

lo%d: can't handle af%d. The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

SEE ALSO

intro(4), inet(4)

BUGS

It should handle all address and protocol families. An approved network address should be reserved for this interface.

NAME

net - network entry device

DESCRIPTION

The special file */dev/net* is opened invisibly to the user the first time the user invokes a Section (3T) or (3W) networking library function. This file is peculiar in that *open(2)* returns the highest available file descriptor. This file descriptor is used to make *ioctl* calls to the network in order to provide the networking services of libraries (3T) and (3W).

FILES

/dev/net

SEE ALSO

intro(3N), intro(3T), intro(3W)

NAME

pty – pseudo terminal driver

SYNOPSIS

pseudo-device pty

DESCRIPTION

The *pty* driver provides support for a device-pair termed a *pseudo terminal*. A pseudo terminal is a pair of character devices, a *master* device and a *slave* device. The slave device provides processes an interface identical to that described in *termio(7)* of the UNIX System V Programmer Reference Manual. However, whereas all other devices which provide the interface described in *termio(7)* have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device.

In configuring, if no optional "count" is given in the specification, 8 pseudo terminal pairs are configured.

The following *ioctl* calls apply only to pseudo terminals:

TIOCSTOP

Stops output to a terminal (e.g. like typing ^S). Takes no parameter.

TIOCSTART

Restarts output (stopped by TIOCSTOP or by typing ^S). Takes no parameter.

TIOCPKT

Enable/disable *packet* mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by

reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent *read* from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

TIOCPKT_FLUSHREAD
: whenever the read queue for the terminal is flushed.

TIOCPKT_FLUSHWRITE
: whenever the write queue for the terminal is flushed.

TIOCPKT_STOP
: whenever output to the terminal is stopped a la ^S.

TIOCPKT_START
: whenever output to the terminal is restarted.

TIOCPKT_DOSTOP
: whenever *t_stopc* is ^S and *t_startc* is ^Q.

TIOCPKT_NOSTOP
: whenever the start and stop characters are not ^S/^Q.

This mode is used by *rlogin(1)* and *rlogind(1M)* to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper backflushing of output; it can be used by other similar programs.

TIOCREMOTE

A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. In normal usage, a write of data is like the data typed as a line on the terminal; a write of "0" bytes is like typing an end-of-file character. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow controlled input is required.

FILES

/dev/pty[p-r][0-9a-f]	master pseudo terminals
/dev/tty[p-r][0-9a-f]	slave pseudo terminals

DIAGNOSTICS

None.

BUGS

It is not possible to send an EOT.

NAME

tcp – Transmission Control Protocol

SYNOPSIS

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

s = socket(AF_INET, SOCK_STREAM, 0);
```

DESCRIPTION

TCP 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". Thus, 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.

SEE ALSO

intro(4), inet(4)

BUGS

It should be possible to send and receive TCP options. The system always tries to negotiate the maximum TCP segment size to be 1024 bytes. This can result in poor performance if an intervening network performs excessive fragmentation.

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

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

s = socket(AF_INET, SOCK_DGRAM, 0);
```

DESCRIPTION

UDP is a simple, unreliable datagram protocol.

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 (i.e. a UDP port may not 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.

SEE ALSO

send(3W), recv(3W), intro(4), inet(4)

NAME

intro – introduction to network data files.

DESCRIPTION

This section describes network data files. These data files are used by Sections 1, 1M, 3, and 3N.

LIST OF FILES

Program	*On Page*	*Description*
.netrc	.netrc.5	info for auto-login validation by *ftp*(1)
.rhosts	.rhosts.5	info for auto-login validation by *rlogind*(1M), *rsh*(1M)
hosts.equiv	hosts.equiv.5	info for auto-login validation by *rlogind*(1M), *rsh*(1M)
ftpusers	ftpusers.5	for *ftp* access validation
localgateway	local.5	for maintenance of local entries
localhosts	local.5	for maintenance of local entries
localnetworks	local.5	for maintenance of local entries
gateways	gateways.5	reformatted information derived from the NIC
hosts	hosts.5	NIC standard host table
networks	networks.5	NIC standard network table
protocols	protocols.5	used by *getprotoent*(3N)
services	services.5	used by *getservent*(3N)

NAME

ftpusers – unacceptable ftp users

DESCRIPTION

Ftpd(1M) checks to see if the login name of the user trying to open a connection is in this file. If it is, *ftpd* denies this user access.

Ftpusers resides in */etc*.

FORMAT

username

.

.

.

username

FILES

/etc/ftpusers

SEE ALSO

ftpd(1M)

NAME

gateways - network gateway information

DESCRIPTION

This file contains information about active and passive *gateways*. It is produced by *htable*(1M) for use by *routed(1M)*.

The file */etc/gateways* is a series of lines, each in the following format:

< net | host > *name1* **gateway** *name2* **metric** *value*
< passive | active >

The **net** or **host** keyword indicates if the route is to a network or specific host.

Name1 is the name of the destination network or host. This may be a symbolic name located in */etc/networks* or */etc/hosts*, or an Internet address specified in "dot" notation; see *inet*(3N).

Name2 is the name or address of the gateway to which messages should be forwarded.

Value is a metric indicating the hop count to the destination host or network.

The keyword **passive** or **active** indicates if the gateway should be treated as *passive* or *active* as described in *routed(1M)*.

FORMAT

< net | host > *name1* **gateway** *name2* **metric** *value*
< passive | active >

SEE ALSO

intro(3W), *htable(1M)*, *routed(1M)*

NAME

hosts.equiv – public information to validate remote auto-login requests

DESCRIPTION

Hosts.equiv is an optional file which, if present, is used by various facilities to authenticate a request for login coming from a user on a remote machine. It contains a list of remote hosts with which the local host has user-names in common. These remote hosts are termed "equivalent" hosts. When a request for login comes from an equivalent host, the remote username being used is sought in the local */etc/passwd* file. If found, the remote user is automatically logged in as himself on the local host. This process takes place invisibly to the remote user.

Normal entries in *hosts.equiv* contain only the name of a remote host. Special entries contain the name of a remote host followed by a username on that host. If, on an incoming request, a special entry is matched but the remote user has supplied an alternate username as the account he would like to use for the remote login, he may do so provided that a) the alternate username is a valid username on the local host, and b) the alternate username does not have super-user privilege.

FORMAT

remotehost

<or>

remotehost trustedremoteuser

Remotehost must be separated from *trustedremoteuser* by a space.

FILES

/etc/passwd
/etc/hosts.equiv

SEE ALSO

rlogin(1), remsh(1), rcmd(3), .rhosts(5), rlogind(1M), rshd(1M)

NAME

hosts – host name data base

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

Internet address
official host name
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet(3N)*. Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts

SEE ALSO

gethostent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast

access.

NAME

localgateways, localhosts, localnetworks – local network entries and aliases

DESCRIPTION

Localgateways, *localhosts*, and *localnetworks* have the same formats as *gateways*, *hosts* and *networks*, respectively. If present, they are added by *htable(1M)* to its output.

SEE ALSO

gateways(4), *hosts(4)*, *networks(4)*, *gettable(1M)*, *htable(1M)*

NAME

.netrc – auto-login information

DESCRIPTION

.netrc contains login information for *ftp(1)* access to a remote machine. When *ftp(1)* is opening a connection to a remote machine, it checks the user's home directory for this file. If it is there, *ftp(1)* checks it for login information for the specified machine. If either *.netrc* does not exist or it exists but contains no entry for the specified machine, the user is prompted for username and password. *.netrc* may contain multiple entries with the proper format, where each entry will enable *ftp(1)* to automatically login to the specified machine.

As a security precaution, *ftp(1)* requires that *.netrc* be read/writeable only by its owner since it contains very sensitive information. Were others allowed access to it, they would have access to a user's login information for all the machines listed in the file.

FORMAT

machine [machine name] login [login name] password [login password]

FILES

~/.netrc

SEE ALSO

ftp(1), rexec(3)

NAME

networks – network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional "." notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

protocols – protocol name data base

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

.rhosts – private information to validate remote auto-login request

DESCRIPTION

.rhosts is an optional file which, if present, is used by various facilities to authenticate a request for login coming from a user on a remote machine. It is essentially a private version of *hosts.equiv(5)*. With it a user may permit specific users on specific remote hosts to automatically login to the local host as himself.

.rhosts is sought and searched only when the remote login request is not matched by an entry in *hosts.equiv(5)*. Each entry in *.rhosts* identifies a remote host and a username on that host. If either one of these fields is not matched by an incoming request, then that entry is not matched. If none of the entries are matched, auto-login is denied and the remote user will be prompted for login and password. If an entry contains only the name of a remote host, the implicit remote username is identical with that of the local user.

.rhosts must reside in the local user's home directory and must be owned by either the local user or root.

FORMAT

remotehost remoteuser

<or>

remotehost

Remotehost must be separated from *remoteuser* by a space.

SEE ALSO

rlogin(1), remsh(1), rcmd(3), rlogind(1M), rshd(1M)

NAME

sendmail.cf – configuration file for WIN/3B mail service

DESCRIPTION

WIN/3B can handle three mail services, *local*, *smtp* and *uucp*. The *sendmail.cf* file is a cryptic set of rules and definitions used by the WIN/3B mail handler to determine how to send a unit of mail. The "to address" entered by a mail user is analysed to determine the mode of delivery. All mail with an ARPA style "to address" is delivered via *smtp*, all mail with a *uucp* style "to address" is delivered via *uucp*, and all mail addressed to a local user is deliverd via */bin/mail*.

WARNING

The *sendmail.cf* file is extremely important for the WIN/3B mail system. It is advised that no modifications be made to this file. Tampering with this file could easily cause the WIN/3B mail system to stop working properly.

SEE ALSO

WIN/3B Administrator Guide.

NAME

services – service name data base

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a "/" is used to separate the port and protocol (e.g. "512/tcp"). A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/services

SEE ALSO

getservent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.