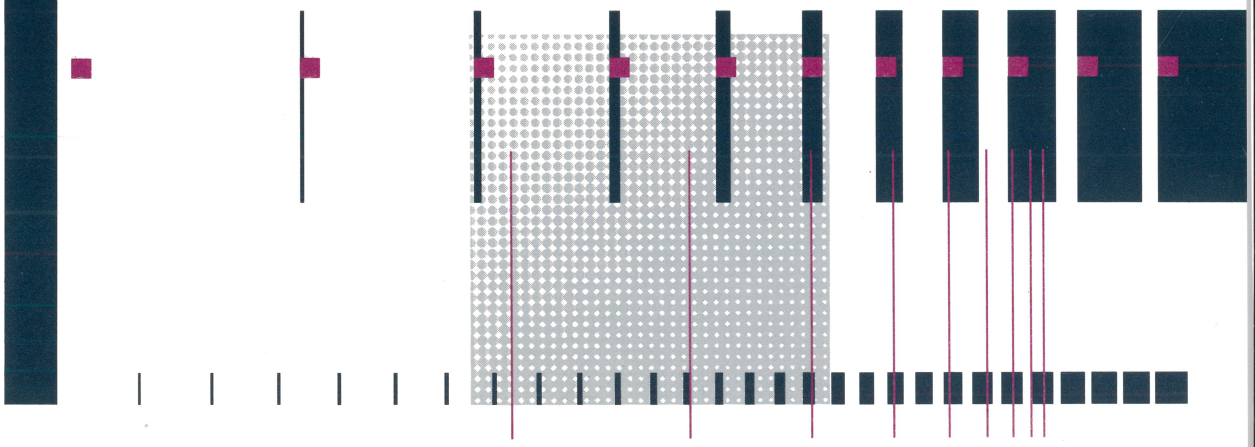
# RISC/os
# System Administrator's
# Reference Manual

*Order Number 3205DOC*



mips

The power of RISC is in the system.

# RISC/os
# System Administrator's
# Reference Manual

*Order Number 3205DOC*

***July 1989***

---

*Your comments on our products and publications are welcome. A postage–paid form is provided for this purpose on the last page of this manual.*

Mfg. Part Number 84–00137–002(B)/02–00134–002

MIPS Computer Systems, Inc.
930 Arques Ave.
Sunnyvale, CA 94086

---

Customer Service Telephone Numbers:

| | | |
|---|---|---|
| California: | (800) | 992–MIPS |
| All other states: | (800) | 443–MIPS |
| International: | (415) | 330–7966 |

Mfg. Part Number 84–00137–002(B)/02–00134–002

## TABLE OF CONTENTS

**1M. System Maintenance**

**7. Special Files**

**8. Special BSD Commands**

## PERMUTED INDEX

**NAME**

Uutry – try to contact remote system with debugging on

**SYNOPSIS**

**/usr/lib/uucp/Uutry** [ **–x** debug_level ] [ **–r** ] system_name

**DESCRIPTION**

*Uutry* is a shell that is used to invoke *uucico* to call a remote site. Debugging is turned on (default is level 5); **–x** will override that value. The **–r** overrides the retry time in **/usr/spool/uucp/.status**. The debugging output is put in file **/tmp/***system_name*. A tail **–f** of the output is executed. A <DELETE> or <BREAK> will give control back to the terminal while the *uucico* continues to run, putting its output in **/tmp/***system_name*.

**FILES**

/usr/lib/uucp/Systems
/usr/lib/uucp/Permissions
/usr/lib/uucp/Devices
/usr/lib/uucp/Maxuuxqts
/usr/lib/uucp/Maxuuscheds
/usr/spool/uucp/*
/usr/spool/locks/LCK*
/usr/spool/uucppublic/*
/tmp/system_name

**SEE ALSO**

uucico(1M),
uucp(1C), uux(1C) in the *User's Reference Manual*.

**NAME**

accept, reject – allow or prevent LP requests

**SYNOPSIS**

**/usr/lib/accept** *destinations*
**/usr/lib/reject** [ **–r**[ *reason* ] ] *destinations*

**DESCRIPTION**

**Accept** allows *lp(1)* To accept requests for the named *destinations*. A *destination* can be either a line printer (LP) or a class of printers. Use *lpstat(1)* to find the status of *destinations*.

**reject** prevents *lp(1)* from accepting requests for the named *destinations*. A *destination* can be either a printer or a class of printers. Use *lpstat*(1) to find the status of *destinations*. The following option is useful with *reject*.

**–r**[ *reason* ] Associates a *reason* with preventing *lp* from accepting requests. This *reason* applies to all printers mentioned up to the next **–r** option. *Reason* is reported by *lp* when users direct requests to the named *destinations* and by *lpstat*(1). If the **–r** option is not present or the **–r** option is given without a *reason*, then a default *reason* will be used.

**FILES**

*/usr/spool/lp/**

**SEE ALSO**

*lpadmin(1M), lpsched(1M).*
*enable(1), lp(1), lpstat(1)* in the *User's Reference Manual.*

**NAME**

arp – address resolution display and control

**SYNOPSIS**

**arp** *hostname*
**arp -a** [ *unix* ] [ *kmem* ]
**arp -d** *hostname*
**arp -s** *hostname ether_addr* [ **temp** ] [ **pub** ] [ **trail** ]
**arp -f** *filename*

**DESCRIPTION**

The **arp** program displays and modifies the Internet-to-Ethernet address translation tables used by the address resolution protocol ( *arp (4p)*).

With no flags, the program displays the current ARP entry for *hostname.* The host may be specified by name or by number, using Internet dot notation. With the **-a** flag, the program displays all of the current ARP entries by reading the table from the file *kmem* (default /dev/kmem) based on the kernel file *unix* (default /unix).

With the **-d** flag, a super-user may delete an entry for the host called *hostname.*

The **-s** flag is given to create an ARP entry for the host called *hostname* with the Ethernet address *ether_addr.* The Ethernet address is given as six hex bytes separated by colons. The entry will be permanent unless the word **temp** is given in the command. If the word **pub** is given, the entry will be "published"; i.e., this system will act as an ARP server, responding to requests for *hostname* even though the host address is not its own. The word **trail** indicates that trailer encapsulations may be sent to this host.

The **-f** flag causes the file *filename* to be read and multiple entries to be set in the ARP tables. Entries in the file should be of the form

*hostname ether_addr* [ **temp** ] [ **pub** ] [ **trail** ]

with argument meanings as given above.

**SEE ALSO**

*inet(3N), arp(7P), ifconfig(1M)*

**NAME**

auto – initiate OS autoboot sequence

**SYNOPSIS**

**auto**

**DESCRIPTION**

The PROM Monitor *auto* command initiates the two-level operating system autoboot sequence. Once initiated, this sequence waits for about 20 seconds. During this delay, you can abort the autoboot sequence by typing a Control-c on the console or you can speed the boot process by pressing the *Enter* key on the keyboard. When the delay expires, the program specified by the PROM Monitor environment variable *bootfile* is loaded and passed the current environment and the argument **-a .**

The default environment variable *bootfile* is "dkip(0,0,8)sash".

**SEE ALSO**

prom(1prom), sash(1Mspp), dvhtool(1)

**NAME**

bfsd – boot file system server

**SYNOPSIS**

**/etc/bfsd −d** *directory* [−u *username* ] [−g *groupname* ] [−i *interface* ]

**DESCRIPTION**

*bfsd* provides remote file access based on the SOCK_DGRAM networking interface to client systems. Typical uses include bringing in bootable images to a diskless network node.

The following command line arguments are supported.

**−d** *directory*
Specifies the directory that the *bfsd* should consider its home. *Bfsd* does a *chdir(2)* to that directory, and uses it as the root for all relative pathnames found in incoming requests.

**−u** *username*
Attempt to run as the specified user. Typically this would be the user that owns the directory specified in the **−d** option.

−g *groupname*
Attempt to run as a member of the specified group. Typically this would be the user that owns the directory specified in the **−d** option.

**−i** *interface*
Specifies a directly connected network interface, such as *enp0.*

**SEE ALSO**

bfs(4P)

**BUGS**

*Bfsd* should be started by the *inetd(8),* rather than operating independently. The packet types related to the data write facility of the protocol are not implemented.

**NAME**

bootp – server for DARPA Bootstrap Protocol (BOOTP)

**SYNOPSIS**

**/usr/etc/bootp** [ **–d** ] [ **–f** ]

**DESCRIPTION**

*Bootp* is a server which supports the DARPA Bootstrap Protocol (BOOTP). This protocol is designed to allow a (possibly diskless) client machine to determine its own Internet address, the address of a boot server and the name of an appropriate boot file to be loaded and executed. BOOTP does not provide the actual transfer of the boot file, which is typically done with a simple file transfer protocol such as TFTP. A detailed protocol specification for BOOTP is contained in RFC 951, which available from the Network Information Center.

The BOOTP protocol uses UDP/IP as its transport mechanism. The BOOTP server receives service requests at the UDP port indicated in the "bootp" service description contained in the file */etc/services* (see *services*(4)). The BOOTP server is started by *inetd*(1M), as configured in the *inetd.conf* file.

The basic operation of the BOOTP protocol is a single packet exchange as follows:

1) The booting client machine broadcasts a BOOTP request packet to the BOOTP server UDP port, using a UDP broadcast or the equivalent thereof. The request packet includes the following information:

   requestor's Ethernet address
   requestor's Internet address (optional)
   desired server's name (optional)
   boot file name (optional)

2) All the BOOTP servers on the same Ethernet wire as the client machine receive the client's request. If the client has specified a particular server, then only that server will respond.

3) The server looks up the requestor in its configuration file by Internet address or Ethernet address, in that order of preference. (The BOOTP configuration file is described below.) If the Internet address was not specified by the requestor and a configuration record is not found, the server will look in the */etc/ethers* file (see *ethers*(4)) for an entry with the client's Ethernet address. If an entry is found, the server will check the hostname of that entry against the */etc/hosts* file (see *hosts*(4)) in order to complete the Ethernet address to Internet address mapping. If the BOOTP request does not include the client's Internet address and the server is unable to translate the client's Ethernet address into an Internet address by either of the two methods described, the server will not respond to the request.

4) The server performs name translation on the boot filename requested and then checks for the presence of that file. If the file is present, then the server will send a response packet to the requestor which includes the following information:

   the requestor's Internet address
   the server's Internet address
   the Internet address of a gateway to the server
   the server's name
   vendor specific information (not defined by the protocol)

   If the boot file is missing, the server will return a response packet with a null filename, but only if the request was specifically directed to that server. The pathname translation is: if the boot filename is rooted, use it as is; else concatenate the root of the

boot subtree, as specified by the BOOTP configuration file, followed by the filename supplied by the requestor, followed by a period and the requestor's hostname. If that file is not present, remove the trailing period and host name and try again. If no boot filename is requested, use the default boot file for that host from the configuration table. If there is no default specified for that host, use the general default boot filename, first with *.hostname* as a suffix and then without.

**Options**

The **−d** option causes *bootp* to generate debugging messages. All messages from *bootp* go through *syslogd*(1M), the system logging daemon.

The **−f** option enables the forwarding function of *bootp*. Refer to the following section on Booting Through Gateways for an explanation.

**Bootp Configuration File**

In order to perform its name translation and address resolution functions, *bootp* requires configuration information, which it gets from an ASCII file called */usr/etc/bootptab* and from other system configuration files like */etc/ethers* and */etc/hosts*. Here is a sample *bootptab* file:

```
#
# /usr/etc/bootptab:  database for bootp server
#
# Blank lines and lines beginning with '#' are ignored.
#
# root of boot subtree

/usr/local/boot

# default bootfile

unix

%%

#
# The remainder of this file contains one line per client interface
# with the information shown by the table headings below.
# The 'host' name is also tried as a suffix for the 'bootfile'
# when searching the boot directory.  (e.g., bootfile.host)
#
# host  htype   haddr           iaddr   bootfile
#

unixbox1        1:2:3:4:bb:cc   89.0.0.2
```

The fields of each line may be separated by variable amounts of white space (blanks and tabs). The first section, up to the line beginning '%%', defines the place where *bootp* looks for boot files when the client requests a boot file using a non-rooted pathname. The second section of the file is used for mapping client Ethernet addresses into Internet addresses. The *htype* field should always have a value of 1 for now, which indicates that the hardware address is a 48-bit Ethernet address. The *haddr* field is the Ethernet address of the system in question expressed as 6 hex bytes separated by colons. The *iaddr* field is the 32-bit Internet address of the system expressed in standard dot notation (4 byte values in decimal, in network order, separated by periods). Each line in the second section can also specify a default boot file for each specific host. In the example above, if the host called *unixbox* makes a BOOTP request

with no boot file specified, the server will select the first of the following that it finds:

/usr/local/boot/unix.unixbox
/usr/local/boot/unix

It is not necessary to create a record for every potential client the every *bootptab* file. The only constraint is that *bootp* will only respond to a request from a client if it can deduce the client's Internet address. There are three ways that this can happen: 1) the client already knows his Internet address and includes it in the BOOTP request packet, 2) there is an entry in */usr/etc/bootptab* that matches the client's Ethernet address or 3) there are entries in the */etc/ethers* and */etc/hosts* files (or their Yellow Pages equivalents) that allow the client's Ethernet address to be translated into an Internet address.

**Booting Through Gateways**

Since the BOOTP request is distributed using a UDP broadcast, it will only be received by other hosts on the same Ethernet cable as the client. In some cases the client may wish to boot from a host on another network. This can be accomplished by using the forwarding function of BOOTP servers on the local wire. To use BOOTP forwarding, there must be a *bootp* process running in a gateway machine on the local cable. A gateway machine is simply a machine with more than one Ethernet controller board. The gateway *bootp* must be invoked with the **-f** option to activate forwarding. Such a forwarding *bootp* will resend any BOOTP request it receives that asks for a specific host by name, if that host is on a different network from the client that sent the request. The BOOTP server forwards the packet using the full routing capabilities of the underlying IP layer in the kernel, so the forwarded packet will automatically be routed to the requested BOOTP server provided that the kernel routing tables contain a route to the destination network.

**DIAGNOSTICS**

The BOOTP server sends any messages it wants to reach the outside world through the system logging daemon, *syslogd*(1M). The actual disposition of these messages depends on the configuration of *syslogd* on the machine in question. Consult *syslogd*(1M) for further information.

*Bootp* can produce the following messages:

'get interface config' ioctl failed (message)
'get interface netmask' ioctl failed (message)
getsockname fails (message)
forwarding failed (message)
send failed (message)
set arp ioctl failed

Each of the above messages mean that a system call has returned an error unexpectedly. Such errors usually cause *bootp* to terminate. The *message* will be the result of calling *perror*(3) with the *errno* value that was returned.

less than two interfaces, -f flag ignored

Warning only. Means that the **-f** option was specified on a machine that is not a gateway. Forwarding only works on gateways.

request for unknown host xxx from yyy

Information only. A BOOTP request was received asking for host *xxx*, but that host is not in the host database. The request was generated by *yyy*, which may be given as a host name or an Internet address.

request from xxx for 'fff'

Information only. *Bootp* logs each request for a boot file. The means that host *xxx* has requested boot file *fff*.

boot file fff missing
: A request has been received for the boot file *fff*, but that file doesn't exist.

replyfile fff
: Information only. *Bootp* has selected the file *fff* as the boot file to satisfy a request.

forward request with gateway address already set (dd.dd.dd.dd)
: The server has received a reply to be forwarded to a requestor, but some other *bootp* has already filled himself in as the gateway. This is an error in the BOOTP forwarding mechanism.

missing gateway address
: This means that this *bootp* has generated a response to a client and is trying to send the response directly to the client (i.e. the request did not get forwarded by another *bootp*), but none of the Ethernet interfaces on this machine is on the same wire as the client machine. This indicates a bug in the BOOTP forwarding mechanism.

can't open /usr/etc/bootptab
: The *bootp* configuration file is missing or has wrong permissions.

(re)reading /usr/etc/bootptab
: Information only. *Bootp* checks the modification date of the configuration file on the receipt of each request and rereads it if it has been modified since the last time it was read.

bad hex address: xxx at line nnn of bootptab
bad internet address: sss at line nnn of bootptab
string truncated: sss, on line nnn of bootptab
: These messages all mean that the format of the BOOTP configuration file is not valid.

'hosts' table length exceeded
: There are too many lines in the second section of the BOOTP configuration file. The current limit is 512.

can't allocate memory
: A call to *malloc*(3) failed.

gethostbyname(sss) fails (message)
: A call to *gethostbyname*(3N) with the argument *sss* has failed.

gethostbyaddr(dd.dd.dd.dd) fails (message)
: A call to *gethostbyaddr*(3N) with the argument *dd.dd.dd.dd* has failed.

can't find source net for address xxx
: This means that the server has received a datagram with a source address that doesn't make sense. The offending address is printed as a 32 bit hexadecimal number *xxx*.

**SEE ALSO**

inetd(1M), syslogd(1M), tftpd(1M), ethers(4), hosts(4), services(4)

NAME

boot – load and execute program

SYNOPSIS

**boot** [ **-f file** ] [ **-n** ] [ *args* ]

DESCRIPTION

*boot* loads the program specified by the **-f** option. If the **-f** flag is not specified, *boot* loads the file specified by the environment variable *bootfile* . If **-n** is specified, *boot* loads the requested file, but does not transfer control to the program. The program can be initiated later using the *go*(1spp) command, but no arguments may be passed in this case. If present *args*, are passed to the program and are accessible from the standard **argc** , **argv** mechanism. Any argument that begins with a **-** must be prepended with an additional **-** , this extra dash will be removed before the argument is passed to the program. The current environment will be passed to the program as the third parameter to the main routine and also from the external variable *environ* .

ENVIRONMENT VARIABLE

If the environment variable $ *path* is defined and the *boot* command has a file to load that does not have a device specification, *boot* tries to load a file name formed by prepending the contents of $ *path* to the original file name. If $ *path* is a list of space separated prefixes, the *boot* command tries each prefix from $ *path* in turn until the file can be successfully booted or all prefixes have been tried.

SEE ALSO

go(1prom), load(1prom), sload(1prom)

**NAME**

brc, bcheckrc – system initialization procedures

**SYNOPSIS**

**/etc/brc**

**/etc/bcheckrc**

**DESCRIPTION**

These shell procedures are executed via entries in */etc/inittab* by *init(1M)* whenever the system is booted (or rebooted).

First, the **bcheckrc** procedure checks the status of the root file system. If the root file system is found to be bad, **bcheckrc** repairs it.

Then, the **brc** procedure clears the mounted file system table, */etc/mnttab* and puts the entry for the root file system into the mount table.

After these two procedures have executed, *init* checks for the *initdefault* value in */etc/inittab*. This tells *init* in which run level to place the system. Since *initdefault* is initially set to **2**, the system will be placed in the multi-user state via the */etc/rc2* procedure.

Note that **bcheckrc** should always be executed before **brc**. Also, these shell procedures may be used for several run-level states.

**SEE ALSO**

*fsck(1M), init(1M), rc2(1M), shutdown(1M).*

**NAME**

captoinfo – convert a termcap description into a terminfo description

**SYNOPSIS**

**captoinfo** [**−v** ...] [**−V**] [**−1**] [**−w** *width*] *file* ...

**DESCRIPTION**

**Captoinfo** looks in *file* for *termcap* descriptions. For each one found, an equivalent *terminfo*(4) description is written to standard output, along with any comments found. A description which is expressed as relative to another description (as specified in the *termcap* *tc*= field) will be reduced to the minimum superset before being output. If no *file* is given, then the environment variable **TERMCAP** is used for the filename or entry. If **TERMCAP** is a full pathname to a file, only the terminal whose name is specified in the environment variable **TERM** is extracted from that file. If the environment variable **TERMCAP** is not set, then the file */etc/termcap* is read.

**−v** print out tracing information on standard error as the program runs. Specifying additional **−v** options will cause more detailed information to be printed.

**−V** print out the version of the program in use on standard error and exit.

**−1** cause the fields to print out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters.

**−w** change the output to *width* characters.

**FILES**

*/usr/lib/terminfo/?/** compiled terminal description database

**CAVEATS**

Certain *termcap* defaults are assumed to be true. For example, the bell character (*terminfo* *bel*) is assumed to be ^*G*. The linefeed capability (*termcap nl*) is assumed to be the same for both *cursor_down* and *scroll_forward* (*terminfo cud1* and *ind*, respectively.) Padding information is assumed to belong at the end of the string. The algorithm used to expand parameterized information for *termcap* fields such as *cursor_position* (*termcap cm*, *terminfo cup*) will sometimes produce a string which, though technically correct, may not be optimal. In particular, the rarely used *termcap* operation **%n** will produce strings that are especially long. Most occurrences of these non-optimal strings will be flagged with a warning message and may need to be recoded by hand. The short two-letter name at the beginning of the list of names in a *termcap* entry, a hold-over from an earlier version of the UNIX system, has been removed.

**DIAGNOSTICS**

tgetent failed with return code n (reason).
The termcap entry is not valid. In particular, check for an invalid 'tc=' entry.

unknown type given for the termcap code *cc*.
The termcap description had an entry for *cc* whose type was not boolean, numeric or string.

wrong type given for the boolean (numeric, string) termcap code *cc*.
The boolean *termcap* entry *cc* was entered as a numeric or string capability.

the boolean (numeric, string) termcap code *cc* is not a valid name.
An unknown *termcap* code was specified.

tgetent failed on TERM=term.
The terminal type specified could not be found in the *termcap* file.

TERM=term: **cap** *cc* (**info** *ii*) is NULL: REMOVED
The *termcap* code was specified as a null string. The correct way to cancel an entry is with an '@', as in ':bs@:'. Giving a null string could cause incorrect

assumptions to be made by the software which uses *termcap* or *terminfo*.

a function key for *cc* was specified, but it already has the value *vv*.
When parsing the **ko** capability, the key *cc* was specified as having the same value as the capability *cc*, but the key *cc* already had a value assigned to it.

the unknown termcap name *cc* was specified in the **ko** termcap capability.
A key was specified in the **ko** capability which could not be handled.

the *vi* character *v* (**info** *ii*) has the value *xx*, but **ma** gives *n*.
The **ma** capability specified a function key with a value different from that specified in another setting of the same key.

the unknown *vi* key *v* was specified in the **ma** termcap capability.
A *vi*(1) key unknown to *captoinfo* was specified in the **ma** capability.

Warning: *termcap* **sg** (*nn*) and *termcap* **ug** (*nn*) had different values.
*terminfo* assumes that the **sg** (now **xmc**) and **ug** values were the same.

Warning: the string produced for *ii* may be inefficient.
The parameterized string being created should be rewritten by hand.

Null termname given.
The terminal type was null. This is given if the environment variable **TERM** is not set or is null.

cannot open *file* for reading.
The specified file could not be opened.

**SEE ALSO**

*infocmp(1M), tic(1M).*
*curses (3X), terminfo(4)* in the *Programmer's Reference Manual.*
Chapter 10 in the *Programmer's Guide*.

**NOTES**

**Captoinfo** should be used to convert *termcap* entries to *terminfo(4)* entries because the *termcap* database (from earlier versions of UNIX System V) may not be supplied in future releases.

**NAME**

cat – display files on console

**SYNOPSIS**

**cat** [*files*]

**DESCRIPTION**

*cat* displays the contents of the listed *files* on the console

**NAME**

**chroot** – change root directory for a command

**SYNOPSIS**

**/etc/chroot** *newroot command*

**DESCRIPTION**

**chroot** causes the given command to be executed relative to the new root. The meaning of any initial slashes (*/*) in the path names is changed for the command and any of its child processes to *newroot*. Furthermore, upon execution, the initial working directory is *newroot*.

Notice, however, that if you redirect the output of the command to a file:

```
chroot newroot command >x
```

will create the file **x** relative to the original root of the command, not the new one.

The new root path name is always relative to the current root: even if a **chroot** is currently in effect, the *newroot* argument is relative to the current root of the running process.

This command can be run only by the super-user.

**SEE ALSO**

*cd(1)* in the *User's Reference Manual.*
*chroot(2)* in the *Programmer's Reference Manual.*

**BUGS**

One should exercise extreme caution when referencing device files in the new root file system.

**NAME**

ckbupscd – check file system backup schedule

**SYNOPSIS**

**/etc/ckbupscd** [ **–m** ]

**DESCRIPTION**

*ckbupscd* consults the file */etc/bupsched* and prints the file system lists from lines with date and time specifications matching the current time. If the **–m** flag is present an introductory message in the output is suppressed so that only the file system lists are printed. Entries in the */etc/bupsched* file are printed under the control of *cron*. The System Administration commands *bupsched/schedcheck* are provided to review and edit the */etc/bupsched* file. The file */etc/bupsched* should contain lines of 4 or more fields, separated by spaces or tabs. The first 3 fields (the schedule fields) specify a range of dates and times. The rest of the fields constitute a list of names of file systems to be printed if **ckbupscd** is run at some time within the range given by the schedule fields. The general format is:

*time*[,*time*] *day*[,*day*] *month*[, *month*] *fsyslist* where:

*time* Specifies an hour of the day (*0* through *23*), matching any time within that hour, or an exact time of day (*0:00* through *23:59*).

*day* Specifies a day of the week (*sun* through *sat*) or day of the month (*1* through *31*).

*month* Specifies the month in which the time and day fields are valid. Legal values are the month numbers (*1* through *12*).

*fsyslist* The rest of the line is taken to be a file system list to print. Multiple time, day, and month specifications may be separated by commas, in which case they are evaluated left to right. An asterisk (*) always matches the current value for that field. A line beginning with a sharp sign (#) is interpretted as a comment and ignored. The longest line allowed (including continuations) is 1024 characters.

**EXAMPLES**

The following are examples of lines which could appear in the */etc/bupsched* file.

06:00-09:00 fri 1,2,3,4,5,6,7,20,9,10,11 /applic
Prints the file system name */applic* if **ckbupscd** is run between 6:00am and 9:00am any Friday during any month except December.

0:00-06:00,16:00-23:59 1,2,3,4,5,6,7 1,20 /
Prints a reminder to backup the root (/) file system if **ckbupscd** is run between the times of 4:00pm and 6:00am during the first week of August or January.

**FILES**

*/etc/bupsched* specification file containing times and file system to back up

**SEE ALSO**

*cron(1M).*
*echo(1), sh(1), sysadm(1)* in the *User's Reference Manual.*

**ERRORS**

**ckbupscd** will report file systems due for backup if invoked any time in the window. It does not know that backups may have just been taken.

**NAME**

clri.ffs – clear i-node

**SYNOPSIS**

**/etc/clri.ffs** filesystem i-number ...

**DESCRIPTION**

**N.B.: clir.ffs** is obsoleted for normal file system repair work by *fsck(1M)*.

**clir.ffs** writes zeros on the i-nodes with the decimal *i-numbers* on the *filesystem*.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

**ERRORS**

If the file is open, **clri.ffs** is likely to be ineffective.

**NAME**

clri.s51k – clear i-node

**SYNOPSIS**

**/etc/clri.s51k** *special i-number ...*

**DESCRIPTION**

**NOTE:** *The obsolete S51K file system has been kept for backward compatibility. The fast file system (FFS) is preferred see fs(4FFS).*

**clri.s51k** writes nulls on the 64 bytes at offset *i-number* from the start of the i-node list. This effectively eliminates the i-node at that address. *Special* is the device name on which a file system has been defined. After **clri.s51k** is executed, any blocks in the affected file will show up as "not accounted for" when *fsck.s51k(1M)* is run against the file-system. The i-node may be allocated to a new file.

Read and write permission is required on the specified *special* device.

This command is used to remove a file which appears in no directory; that is, to get rid of a file which cannot be removed with the *rm* command.

**SEE ALSO**

*fsck.s51k(1M), fsdb.s51k(1M), ncheck.s51k(1M).*
*fs(4S51K)* in the *Programmer's Reference Manual.*
*rm(1)* in the *User's Reference Manual.*

**WARNINGS**

If the file is open for writing, **clri.s51k** will not work. The file system containing the file should be NOT mounted.

If **clri.s51k** is used on the i-node number of a file that does appear in a directory, it is imperative to remove the entry in the directory at once, since the i-node may be allocated to a new file. The old directory entry, if not removed, continues to point to the same file. This sounds like a link, but does not work like one. Removing the old entry destroys the new file.

**NAME**

crash – examine system images

**SYNOPSIS**

**/etc/crash** [ **–d** *dumpfile* ] [ **–n** *namelist* ] [ **–w** *outputfile* ]

**DESCRIPTION**

The *crash* command is used to examine the system memory image of a live or a crashed system by formatting and printing control structures, tables, and other information. Command line arguments to *crash* are *dumpfile, namelist*, and *outputfile*.

*dumpfile* is the file containing the system memory image. The default *dumpfile* is */dev/mem*. The system image can also be the pathname of a file (vmcore.*) produced by *savecore*(1M).

The text file *namelist* contains the symbol table information needed for symbolic access to the system memory image to be examined. The default *namelist* is */unix*. If a system image from another machine is to be examined, the corresponding text file must be copied from that machine.

When the *crash* command is invoked, a session is initiated. The output from a *crash* session is directed to *outputfile*. The default *outputfile* is the standard output.

Input during a *crash* session is of the form:

function [ argument ... ]

where *function* is one of the *crash* functions described in the "FUNCTIONS" section of this manual page, and *arguments* are qualifying data that indicate which items of the system image are to be printed.

The default for process-related items is the current process for a running system and the process that was running at the time of the crash for a crashed system. If the contents of a table are being dumped, the default is all active table entries.

The following function options are available to *crash* functions wherever they are semantically valid.

| | |
|---|---|
| **–e** | Display every entry in a table. |
| **–f** | Display the full structure. |
| **–p** | Interpret all address arguments in the command line as *physical* addresses. |
| **–s** process | Specify a process slot other than the default. |
| **–w** file | Redirect the output of a function to *file*. |

Note that if the **–p** option is used, all address and symbol arguments explicitly entered on the command line will be interpreted as physical addresses. If they are not physical addresses, results will be inconsistent.

The functions *mode, defproc*, and *redirect* correspond to the function options **–p**, **–s**, and **–w**. The *mode* function may be used to set the address translation mode to physical or virtual for all subsequently entered functions; *defproc* sets the value of the process slot argument for subsequent functions; and *redirect* redirects all subsequent output.

Output from *crash* functions may be piped to another program in the following way:

function [ argument ... ] ! shell_command

For example,

**mount ! grep rw**

will write all mount table entries with an *rw* flag to the standard output. The redirection option (**−w**) cannot be used with this feature.

Depending on the context of the function, numeric arguments will be assumed to be in a specific radix. Counts are assumed to be decimal. Addresses are always hexadecimal. Table address arguments larger than the size of the function table will be interpreted as hexadecimal addresses; those smaller will be assumed to be decimal slots in the table. Default bases on all arguments may be overridden. The C conventions for designating the bases of numbers are recognized. A number that is usually interpreted as decimal will be interpreted as hexadecimal if it is preceded by *0x* and as octal if it is preceded by *0*. Decimal override is designated by *0d*, and binary by *0b*.

Aliases for functions may be any uniquely identifiable initial substring of the function name. Traditional aliases of one letter, such as *p* for *proc*, remain valid.

Many functions accept different forms of entry for the same argument. Requests for table information will accept a table entry number, a physical address, a virtual address, a symbol, a range, or an expression. A range of slot numbers may be specified in the form *a−b* where *a* and *b* are decimal numbers. An expression consists of two operands and an operator. An operand may be an address, a symbol, or a number; the operator may be +, −, *, /, &, or |. An operand which is a number should be preceded by a radix prefix if it is not a decimal number (*0* for octal, *0x* for hexidecimal, *0b* for binary). The expression must be enclosed in parentheses ( ). Other functions will accept any of these argument forms that are meaningful.

Two abbreviated arguments to *crash* functions are used throughout. Both accept data entered in several forms. They may be expanded into the following:

table_entry = table entry | address | symbol | range | expression

start_addr = address | symbol | expression

## FUNCTIONS

**?** [ **−w** file ] List available functions.

**!cmd** Escape to the shell to execute a command.

**adv** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the advertise table.

**base** [ **−w** file ] number ...
Print *number* in binary, octal, decimal, and hexadecimal. A number in a radix other then decimal should be preceded by a prefix that indicates its radix as follows: *0x*, hexidecimal; *0*, octal; and *0b*, binary.

**buffer** [ **−w** file] [ −format ] bufferslot

or

**buffer** [ **−w** file] [ −format ] [ **−p** ] start_addr
Alias: **b**.
Print the contents of a buffer in the designated format. The following format designations are recognized: **−b**, byte: **−c**, character; **−d**, decimal; **−x**, hexadecimal; **−o**, octal; **−r**, directory; and **−i**, inode. If no format is given, the previous format is used. The default format at the beginning of a *crash* session is hexadecimal.

**bufhdr** [ **−f** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Alias: **buf**.
Print system buffer headers.

**callout** [ **−w** file ]
Alias: **c**.
Print the callout table.

**dballoc** [ **−w** file ] [ class ... ]
Print the dballoc table. If a class is entered, only data block allocation information for that class will be printed.

**dbfree** [ **−w** file ] [ class ... ]
Print free streams data block headers. If a class is entered, only data block headers for the class specified will be printed.

**dblock** [ **−e** ] [ **−w** file ] [ **−c** class ... ]

or

**dblock** [ **−e** ] [ **−w** file ] [[ **−p** ] table_entry ... ]
Print allocated streams data block headers. If the class option (**−c**) is used, only data block headers for the class specified will be printed.

**defproc** [ **−w** file ] [ **−c** ]

or

**defproc** [ **−w** file ] [ slot ]
Set the value of the process slot argument. The process slot argument may be set to the current slot number (**−c**) or the slot number may be specified. If no argument is entered, the value of the previously set slot number is printed. At the start of a *crash* session, the process slot is set to the current process.

**dis** [ **−w** file ] [ **−a** ] [ **−h** ] start_addr [ count ]
Disassemble from the start address for *count* instructions. The default count is 1. The absolute option (**−a**) specifies a non-symbolic disassembly. The option **−h** means. print register hardware names instead of register compiler names.

**ds** [ **−w** file ] virtual_address ...
Print the data symbol whose address is closest to, but not greater than, the address entered.

**file** [ **−e** ] [ **−w** file ] [[ **−p** ] table_entry ... ]
Alias: **f.**
Print the file table.

**findaddr** [ **−w** file ] table slot
Print the address of *slot* in *table*. Only tables available to the *size* function are available to *findaddr*.

**findslot** [ **−w** file ] virtual_address ...
Print the table, entry slot number, and offset for the address entered. Only tables available to the *size* function are available to *findslot*.

**fs** [ **−w** file ] [[ **−p** ] table_entry ... ]
Print the file system information table.

**gdp** [ **−e** ] [ **−f** ] [ **−w** file ] [[ **−p** ] table_entry ... ]
Print the gift descriptor protocol table.

**help** [ **−w** file ] function ...
Print a description of the named function, including syntax and aliases.

**inode** [ **−e** ] [ **−f** ] [ **−w** file ] [[ **−p** ] table_entry ... ]
Alias: **i.**
Print the inode table, including file system switch information.

**lck** [ **−e** ] [ **−w** file ] [[ **−p** ] table_entry ... ]
Alias: **l.**
Print record locking information. If the **−e** option is used or table address arguments

are given, the record lock list is printed. If no argument is entered, information on locks relative to inodes is printed.

**linkblk** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the linkblk table.

**major** [ **−w** file ] [ entry ... ]
Print the MAJOR table.

**map** [ **−w** file ] mapname ...
Print the map structure of the given mapname.

**mbfree** [ **−w** file ]
Print free streams message block headers.

**mblock** [ **−e** ] [ **−w** filename ] [ [ **−p** ] table_entry ... ]
Print allocated streams message block headers.

**mode** [ **−w** file ] [ mode ]
Set address translation of arguments to virtual (**v**) or physical (**p**) mode. If no mode argument is given, the current mode is printed. At the start of a *crash* session, the mode is virtual.

**mount** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Alias: **m**.
Print the mount table.

**nm** [ **−w** file ] symbol ...
Print value and type for the given symbol.

**od** [ **−p** ] [ **−w** file ] [ −format ] [ −mode ] [ **−s** process ] start_addr [ count ]
Alias: **rd**.
Print *count* values starting at the start address in one of the following formats: character (**−c**), decimal (**−d**), hexadecimal (**−x**), octal (**−o**), ascii (**−a**), or hexadecimal/character (**−h**), and one of the following modes: long (**−l**), short (**−t**), or byte (**−b**). The default mode for character and ascii formats is byte; the default mode for decimal, hexadecimal, and octal formats is long. The format **−h** prints both hexadecimal and character representations of the addresses dumped; no mode needs to be specified. When format or mode is omitted, the previous value is used. At the start of a *crash* session, the format is hexadecimal and the mode is long. If no count is entered, 1 is assumed.

**pcb** [ **−w** file ] [ process ]
Print the process control block. If no arguments are given, the pcb for the current process is printed.

**pdt** [ **−e** ] [ **−w** file ] [ **−s** process ] uvaddr [ count ]

or

**pdt** [ **−e** ] [ **−w** file ] [ **−s** process ] [ **−p** ] start_addr [ count ]
The page descriptor table of the segment which includes the user virtual (KUSEG) address *uvaddr* is printed. Alternatively, the page descriptor table starting at the start address for *count* entries is printed. If no count is entered, 512 (NPGPT) is assumed.

**pfdat** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the pfdata table.

**proc** [**−f**] [ **−w** file ] [ [ **−p** ] table_entry ... #procid ... ]

or

**proc** [**−f**] [ **−w** file ] [ **−r** ]

Alias: **p**.
Print the process table. Process table information may be specified in two ways. First, any mixture of table entries and process ids may be entered. Each process id must be preceded by a **#**. Alternatively, process table information for runnable processes may be specified with the runnable option (**−r**).

**qrun** [ **−w** file ]
Print the list of scheduled streams queues.

**queue** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print streams queues.

**quit** Alias: **q**.
Terminate the *crash* session.

**rcvd** [ **−e** ] [ **−f** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the receive descriptor table.

**redirect** [ **−w** file ] [ **−c** ]

or

**redirect** [ **−w** file ] [ file ]
Used with a file name, redirects output of a *crash* session to the named file. If no argument is given, the file name to which output is being redirected is printed. Alternatively, the close option (**−c**) closes the previously set file and redirects output to the standard output.

**region** [**−e** ] [ **−f** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the region table.

**search** [ **−p** ] [ **−w** file ] [ **−m** mask ] [ **−s** process ] pattern start_addr length
Print the words in memory that match *pattern*, beginning at the start address for *length* words. The mask is anded (&) with each memory word and the result compared against the pattern. The mask defaults to 0xffffffff.

**size** [ **−w** file ] [ **−x** ] [ structure_name ... ]
Print the size of the designated structure. The (**−x**) option prints the size in hexadecimal. If no argument is given, a list of the structure names for which sizes are available is printed.

**sndd** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the send descriptor table.

**srmount** [ **−e** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the server mount table.

**stack** [ **−w** file ] [ **−u** ] [ process ]

or

**stack** [ **−w** file ] [ **−k** ] [ process ]
Alias: **s**.
Dump stack. The (**−u**) option prints the user stack. The (**−k**) option prints the kernel stack. If no arguments are entered, the kernel stack for the current process is printed.

**stat** [ **−w** file ]
Print system statistics and the putbuf array, which contains the latest messages printed via the kernel printf/cmn_err routines.

**stream** [ **−e** ] [ **−f** ] [ **−w** file ] [ [ **−p** ] table_entry ... ]
Print the streams table.

**strstat** [ **−w** file ]
Print streams statistics.

**trace** [ **−w** file ] [ **−r** ] [ process ]
or

**trace** [ **−w** file ] [ **−s** ] [ process ]
or

**trace** [ **−w** file ] [ **−a** ] [ process ]
Alias: **t.**
Print kernel stack trace. The pcb values for sp and pc are used with the **−s** option. For **-r**, *crash* looks for a stack trace of maximal length in the system stack using some heuristics. If these heuristics lead to an "impossible" stack trace, other, shorter traces can be tried with the **-a** option. If none of the **-[rsa]** options is given, **-s** is used for sleeping processes (SSLEEP or SXBRK), and **-r** for running processes.

**ts** [**−w** file ] virtual_address ...
Print closest text symbol to the designated address.

**user** [ **−f** ] [ **−w** file ] [ process ]
Alias: **u.**
Print the ublock for the designated process.

**var** [ **−w** file ]
Alias: **v.**
Print the tunable system parameters.

**vtop** [ **−w** file ] [ **−s** process ] start_addr ...
Print the physical address translation of the virtual start address.

**FILES**

/dev/mem system image of currently running system

**SEE ALSO**

savecore(1M).

**NAME**

cron – clock daemon

**SYNOPSIS**

**/etc/cron**

**DESCRIPTION**

**cron** executes commands at specified dates and times. Regularly scheduled commands can be specified according to instructions found in *crontab* files in the directory */usr/spool/cron/crontabs*. A special file, called **periodic**, owned by **root** (but not the **root** *crontab* file), is also located in the */usr/spool/cron/crontabs* directory. This file is intended to run periodic commands on behalf of the kernel rather than the **root** user. It must not be modified or deleted, nor can it be submitted via the *crontab(1)* command; it is started when *cron* is initialized. Users can submit their own *crontab* file via the *crontab(1)* command. Commands which are to be executed only once may be submitted via the *at(1)* command.

**cron** only examines *crontab* files and *at* command files during process initialization and when a file changes via *crontab* or *at*. This reduces the overhead of checking for new or changed files at regularly scheduled intervals.

Since **cron** never exits, it should be executed only once. This is done routinely through */etc/rc2.d/S75cron* at system boot time. */usr/lib/cron/FIFO* is used as a lock file to prevent the execution of more than one **cron**.

**FILES**

| | |
|---|---|
| */usr/lib/cron* | main cron directory |
| */usr/lib/cron/FIFO* | used as a lock file |
| */usr/lib/cron/log* | accounting information |
| */usr/spool/cron* | spool area |
| */usr/spool/cron/crontabs/periodic* | special **root** file |

**SEE ALSO**

*at(1), crontab(1), sh(1)* in the *User's Reference Manual*.

**DIAGNOSTICS**

A history of all actions taken by **cron** are recorded in */usr/lib/cron/log*.

**NAME**

dd – convert and copy a file

**SYNOPSIS**

**dd** [*option=value*] ...

**DESCRIPTION**

**dd** copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

| *option* | *values* |
|---|---|
| **if=***file* | input file name; standard input is default |
| **of=***file* | output file name; standard output is default |
| **ibs=***n* | input block size *n* bytes (default 512) |
| **obs=***n* | output block size (default 512) |
| **bs=***n* | set both input and output block size, superseding *ibs* and *obs*; also, if no conversion is specified, it is particularly efficient since no in-core copy need be done |
| **cbs=***n* | conversion buffer size |
| **skip=***n* | skip *n* input blocks before starting copy |
| **seek=***n* | seek *n* blocks from beginning of output file before copying |
| **count=***n* | copy only *n* input blocks |
| **conv=ascii** | convert EBCDIC to ASCII |
| **ebcdic** | convert ASCII to EBCDIC |
| **ibm** | slightly different map of ASCII to EBCDIC |
| **block** | Convert variable length records to fixed length |
| **unblock** | Convert fixed length records to fixed variable |
| **lcase** | map alphabetics to lower case |
| **ucase** | map alphabetics to upper case |
| **swab** | swap every pair of bytes |
| **noerror** | do not stop processing on an error |
| **sync** | pad every input block to *ibs* |
| ..., ... | several comma-separated conversions |

Where sizes are specified, a number of bytes is expected. A number may end with **k**, **b**, or **w** to specify multiplication by 1024, 512, or 2, respectively; a pair of numbers may be separated by **x** to indicate multiplication.

*Cbs* is used only if *ascii*, *unblock*, *ebcdic*, *ibm*, or *block* are specified. In the first two cases, *cbs* characters are placed into the conversion buffer (converted to ASCII). Trailing blanks are trimmed and a new-line added before sending the line to the output. In the other cases, ASCII characters are read into the conversion buffer (converted to EBCDIC). Blanks are added to make up an output block of size *cbs*.

After completion, **dd** reports the number of whole and partial input and output blocks.

**DIAGNOSTICS**

| | |
|---|---|
| *f+p blocks in(out)* | numbers of full and partial blocks read(written) |

**NAME**

devinfo – print device specific information

**SYNOPSIS**

**/usr/lbin/devinfo -p** *special* **/usr/lbin/devinfo -i** *special*

**DESCRIPTION**

The *devinfo* command is used to print device specific information about disk devices on standard out.

The options have the following effect:

**–i** option will print the following device information:

| | |
|---|---|
| **Device name** | **Software version** |
| **Drive identification number** | **Device blocks per cylinder** |
| **Device bytes per block** | **Number of device partitions with a block size greater than zero** |

**–p** will print the following device partition information:

| | |
|---|---|
| **Device name** | **Device major and minor numbers** |
| **Partition start block** | **Number of blocks allocated to the partition** |
| **Partition flag** | **Partition tag** |

The command is used by various other commands to obtain device specific information for the making of file systems and determining partition information.

**SEE ALSO**

*prtvtoc(1M).*

**NAME**

devnm – device name

**SYNOPSIS**

**/etc/devnm** [ *names* ]

**DESCRIPTION**

*devenm* identifies the special file associated with the mounted file system where the argument *name* resides.

This command is most commonly used by */etc/brc* (see *brc(1M)*) to construct a mount table entry for the *root* device.

**EXAMPLE**

The command:

```
/etc/devnm /usr
```

produces

```
/dev/dsk/c1d0s2 usr
```

if */usr* is mounted on */dev/dsk/c1d0s2*.

**FILES**

*/dev/dsk/**
*/etc/mnttab*

**SEE ALSO**

*brc(1M)*.

**NAME**

devstr - print device strings

**SYNOPSIS**

**/etc/devstr** [ **−a** ] [ **−f** *format device...*

**DESCRIPTION**

The command **devstr** obtains the device identifier string from each named device, and prints the information according to the format.

As with programs like *date(1)* and *uptime(1),* the format string can contain any text. The sequences \n, \t, \f, \r, \b, and \\ are handled just like C escapes. All other escaped characters are printed as-is. Statistics are printed by using %-specifiers, as in the *date(1)* command. The available specifiers are:

| | |
|---|---|
| f | The name of the device |
| v | The vendor name |
| V | The vendor name padded to 8 characters |
| v | The product id |
| V | The product id padded to 16 characters |
| v | The revision number |
| V | The revision number padded to 4 characters |
| % | The character % |

The padded items are provided because the actual device strings are padded in this way, thus the original device strings can be recreated.

The default format is: %V-%P-%R.

**OPTIONS**

**−a** Use the special alternate format: %f %V-%P-%R. Note that the last option on the command line is the one used.

**−f** *format*
Use the specified format. Note that the last option on the command line is the one used.

**SEE ALSO**

*uname(1).*

**NAME**

df – report number of free disk blocks and inodes

**SYNOPSIS**

**df** [ **–b** ] [ **–f** ] [ **–i** ] [ **–t** *types* ] [ **–u** ] [ **–k** ] [ **–q** ] [ *name ...* ]

**DESCRIPTION**

*df* reports disk space usage statistics for the named filesystems or for all mounted filesystems if none are named. *name* may be any filename that corresponds to the desired filesystem: the disk device, the root of the filesystem, or any directory or file in that filesystem. The information supplied is the name of the filesystem, the type (ffs, nfs, s51k, etc.), the number of kbytes available in the partition, the number of kbytes in use, the number of kbytes free in the filesystem, the percentage of the available space that is free, and the name of the root of the filesystem. In addition, if the **–i** option is given, the number of inodes available, used, free, and the percentage of free inodes are also listed. **NOTE:** This version of prints in a Sun/BSD-like format since the system uses filesystems from these systems.

**OPTIONS**

| | |
|---|---|
| **–b** | Print statistics in 512-byte units. The default is to use 10220-byte units. |
| **–f** | Scan the free list instead of trusting the values given back by the *statfs(2)* system call. |
| **–i** | Print statistics about inodes in addition to file space. |
| **–t** *types* | Print information about filesystems only if they match the named *types*. The *types* argument consists of a comma-separated list of filesystem type names, such as **nfs**, **ffs**, and **s51k**. In addition, the special type **local** matches any non-NFS filesystem. Multiple **–t** options may be given. |
| **–u** | Verify that the host associated with an NFS filesystem is up before attempting to obtain statistics. This option is not yet implemented, and will not be if NFS is changed to not hang for down hosts. |
| **–k** | Not used. Recognized for compatability with other systems. |
| **–q** | Not used. Recognized for compatability with other systems. |

**FILES**

*/etc/fstab*
*/etc/mtab*

**SEE ALSO**

*statfs(2), mntent(20).*

**ERRORS**

Inode counts for NFS entries are always 0 because the NFS protocol does not supply this information. An empty filesystem is reported as being partially (usually 10%) full. This is due to the fact that the free space reported by the operating system is the amount of space that can be used by a non-superuser, whereas the available space reported is the total amount of space on the partition.

**NAME**

du – summarize disk usage

**SYNOPSIS**

**du** [ **–L** ] [ **–s** ] [ **–a** ] [ **–r** ] [ *name...* ]

**DESCRIPTION**

*du* reports the number of blocks contained in all files and (recursively) directories within each directory and file specified by the *names* argument. The block count includes the indirect blocks of the file. If *names* is missing, the current directory is used.

The optional arguments are as follows:

**–s** causes only the grand total (for each of the specified *names*) to be given.

**–a** causes an output line to be generated for each file.

If neither **–s** or **–a** is specified, an output line is generated for each directory only. If both are specified, the last one specified on the command line is used.

**–r** will cause **du** to generate messages about directories that cannot be read, files that cannot be opened, etc., rather than being silent (the default).

**–L** causes **du** to follow symbolic links. Note that this can result in looping if the symbolic link points to a parent of the directory containing the link.

A file with two or more links is only counted once.

**BUGS**

If the **–a** option is not used, non-directories given as arguments are not listed.

Files with holes in them will get an incorrect block count. (See Chapter 5, File System Administration, in the *System Administrator's Guide*)

**NAME**

dump.ffs – incremental file system dump

**SYNOPSIS**

**/etc/dump.ffs** [ *key* [ *argument ...* ] *filesystem* ]

**DESCRIPTION**

*dump.ffs* copies to magnetic tape all files changed after a certain date in the *filesystem*. The *key* specifies the date and other options about the dump. *key* consists of characters from the set **0123456789fiusdWnb.**

**0–9** This number is the "dump level." All files modified since the last date stored in the file */etc/dumpdates* for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option **0** causes the entire filesystem to be dumped.

**f** Place the dump on the next *argument* file instead of the tape. If the name of the file is "–", **dump.ffs** writes to standard output.

**u** If the dump completes successfully, write the date of the beginning of the dump on file */etc/dumpdates*. This file records a separate date for each filesystem and each dump level. The format of */etc/dumpdates* is readable by people, consisting of one free format record per line: filesystem name, increment level and *ctime(3C)* format dump date. */etc/dumpdates* may be edited to change any of the fields, if necessary.

**s** The size of the dump tape is specified in feet. The number of feet is taken from the next *argument*. When the specified size is reached, **dump.ffs** will wait for reels to be changed. The default tape size is 2300 feet.

**d** The density of the tape, expressed in BPI, is taken from the next *argument*. This is used in calculating the amount of tape used per reel. The default is 1600.

**W** *dump.ffs* tells the operator what file systems need to be dumped. This information is gleaned from the files */etc/dumpdates* and */etc/fstab*. The **W** option causes **dump.ffs** to print out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped. If the **W** option is set, all other options are ignored, and **dump.ffs** exits immediately.

**w** Is like **W**, but prints only those filesystems which need to be dumped.

**i** Exit after displaying the number of blocks that should be dumped.

**n** Whenever **dump.ffs** requires operator attention, notify by means similar to a *wall*(1) all of the operators in the group "operator".

**b** The number of dump records per tape block is taken from the next argument. By default, low density (less than 6250 bpi) uses 10 records per block and high density uses 32.

If no arguments are given, the *key* is assumed to be **9u** and a default file system is dumped to the default tape.

*dump.ffs* requires operator intervention on these conditions: end of tape, end of dump, tape write error, tape open error or disk read error (if there are more than a threshold of 32). In addition to alerting all operators implied by the **n** key, **dump.ffs** interacts with the operator on **dump.ffs**'s control terminal at times when **dump.ffs** can no longer proceed, or if something is

grossly wrong. All questions **dump.ffs** poses *must* be answered by typing "yes" or "no", appropriately.

Since making a dump involves a lot of time and effort for full dumps, **dump.ffs** checkpoints itself at the start of each tape volume. If writing that volume fails for some reason, **dump.ffs** will, with operator permission, restart itself from the checkpoint after the old tape has been rewound and removed, and a new tape has been mounted.

*dump.ffs* tells the operator what is going on at periodic intervals, including usually low estimates of the number of blocks to write, the number of tapes it will take, the time to completion, and the time to the tape change. The output is verbose, so that others know that the terminal controlling **dump.ffs** is busy, and will be for some time.

Now a short suggestion on how to perform dumps. Start with a full level 0 dump

dump.ffs 0un

Next, dumps of active file systems are taken on a daily basis, using a modified Tower of Hanoi algorithm, with this sequence of dump levels:

3 2 5 4 7 6 9 8 9 ...

For the daily dumps, a set of 10 tapes per dumped file system is used on a cyclical basis. Each week, a level 1 dump is taken, and the daily Hanoi sequence repeats with 3. For weekly dumps, a set of 5 tapes per dumped file system is used, also on a cyclical basis. Each month, a level 0 dump is taken on a set of fresh tapes that is saved forever.

**FILES**

| | |
|---|---|
| */dev/usr* | default filesystem to dump from |
| */dev/mt/ctape0* | default tape unit to dump to |
| */etc/dumpdates* | new format dump date record |
| */etc/fstab* | dump table: file systems and frequency |
| */etc/group* | to find group *operator* |

**SEE ALSO**

*restore(1FFS), fstab(4)*

**DIAGNOSTICS**

Many, and verbose.

*dump.ffs* exits with zero status on success. Startup errors are indicated with an exit code of 1; abnormal termination is indicated with an exit code of 3.

**ERRORS**

Fewer than 32 read errors on the filesystem are ignored. Each reel requires a new process, so parent processes for reels already written just hang around until the entire tape is written.

*dump.ffs* with the **W** or **w** options does not report filesystems that have never been recorded in */etc/dumpdates*, even if listed in */etc/fstab*.

It would be nice if **dump.ffs** knew about the dump sequence, kept track of the tapes scribbled on, told the operator which tape to mount when, and provided more assistance for the operator running *restore*.

Quarter-inch cartridge (QIC) tapes do not have a specific tape density. The default density and length (1600bpi and 2300 ft.) seem to work very well for these tapes.

dump 0unfds /dev/rmt/m0 23000 570

**NAME**

dump – front-end for filesystem dump command

**SYNOPSIS**

**/etc/dump** [ */etc/dump.ffs arguments* ]

**DESCRIPTION**

This command is a front-end program that executes the command */etc/dump.ffs* if all of the named filesystems (or */dev/root* by default) are ffs filesystems.

The key options **f**, **s**, **b**, and **d** are checked for corresponding arguments. If the **W** or **w** keys are given, no filesystem checking is done.

**SEE ALSO**

*dump(1FFS)*.

**NAME**

dump – display contents of memory

**SYNOPSIS**

**dump** [ **-Bcdoux** ] [ **-bhw** ] *range*

**DESCRIPTION**

*dump* formats and displays the contents of memory. You can display the contents of memory in hexadecimal, octal, decimal, unsigned decimal, ASCII, or binary. The contents of memory can be dumped in byte, halfword or word size units.

The default format is hexadecimal ( **-x** ). You can select an alternative format by entering one of the following characters on the command line as an argument.

**-B** Binary format

**-c** ASCII character format

**-d** Decimal format

**-o** Octal format

**-u** Unsigned decimal

**-x** Hex format

The default width is word (32 bits). An alternate width can be selected by one of the following characters on the command line as an argument:

**-b** Byte (8 bits)

**-h** Halfword (16 bits)

**-w** Word (32 bits)

The range specification indicates the amount of memory to be displayed. You can specify the range in one of the following ways:

| | |
|---|---|
| base | Display the contents of the memory address at *base*. |
| base#count | Display the contents of memory starting at *base* and ending at *base* + *count* . |
| base:limit | Displays the contents of the memory addresses starting at *base* and ending at *limit*. |

**EXAMPLE**

The following example shows a base#count range specified in halfwords. The default for the format of hexadecimal is used because no argument was specified. The specified range is displayed on the screen horizontally.

```
>>dump -h 0xbfc04000#5

0xbfc04000: 8dce  514   6  6900  193
```

**SEE ALSO**

g(1prom), p(1prom), fill(1prom)

**NAME**

dumpfs – dump file system information

**SYNOPSIS**

**dumpfs** *filesys* | *device*

**DESCRIPTION**

*dumpfs* prints out the super block and cylinder group information for the file system or special device specified. The listing is very long and detailed. This command is useful mostly for finding out certain file system information such as the file system block size and minimum free space percentage.

**SEE ALSO**

*fs(1FFS), fsck(1FFS), newfs(1FFS), tunefs(1FFS).*
*disktab(5)* in the *Programmer's Reference Manual.*

**NAME**

dvhtool – command to modify disk volume header information

**SYNOPSIS**

**/etc/dvhtool** [**-p** [**modify** *part nblks 1st_blk type*] [*list*] ]
[**-v** [**add** *unix_file dvh_file*] [**creat** *unix_file dvh_file*]
[**delete** *dvh_file*] [*list*]]
[**-d** [**modify** *name value*] [**list**]]

**DESCRIPTION**

*dvhtool* allows modification of the disk volume header information, a block located at the beginning of all disk media. The disk volume header consists of three main parts: the device parameters, the partition table, and the volume directory. The volume directory is used to locate such things as the boot block and the bad sector table. The partition table describes the logical device partitions. The device parameters describe the specifics of a particular disk drive.

Invoked with no arguments, *dvhtool* allows the user to interactively examine and modify the disk volume header. The *read* command reads the volume header from the specified device, usually */dev/rip0vol*. The *vd, pt,* and *dp* commands first list their respective portions of the volume header and them prompt for modifications. The *write* command writes the possibly modified volume header to the device.

Invoked with arguments, **dvhtool** reads the volume header, performs the specified operations, and then writes the volume header. The following describes *dvhtool*'s command line arguments.

The **-v** flag provides four options for modifying the volume directory information in the disk volume header. The **creat** option allows creation of a volume directory entry with the name *dvh_file* and the contents of *unix_file*. If an entry already exists with the name *dvh_file,* it is overwritten with the new contents. The **add** option adds a volume directory entry with the name *dvh_file* and the contents of *unix_file*. Unlike the **creat** option, the **add** options will not overwrite an existing entry. The **delete** option removes the entry named *dvh_file,* if it exists, from the volume directory. The **list** option lists the current volume directory contents.

The **-p** flag provides two options for modifying the partition table information in the disk volume header. The **modify** option allows modification of the partition entry number specified by *part*. The number of blocks in the partition, the first logical block number in the partition, and the partition type are set as specified. The **list** option lists the current partition table contents.

The **-d** flag provides two options for modifying the device parameter information in the disk volume header. The **modify** option sets the *name* device parameter to the specified *value*. The **list** option lists the current device parameters.

**SEE ALSO**

*dkip(7)*

**ERRORS**

Only the command line options for manipulating the volume directory are currently supported.

NAME

enable, disable – enable and disable console devices

SYNOPSIS

**enable** [*console_dev*]
**disable** [*console_dev*]

DESCRIPTION

*enable* allows input from and output to the specified console device from the PROM Monitor, standalone shell, and the debug monitor. *disable* does not allow input from and output to the specified console device. If you use *enable* or *disable* without arguments, then the current set of enabled console devices is displayed.

SEE ALSO

tty(4spp)

**NAME**

ff.1s51k – list file names and statistics for a file system

**SYNOPSIS**

**/etc/ff.S51K** [ *options* ] *special*

**DESCRIPTION**

**NOTE:** *The obsolete S51K file system has been kept for backward compatibility. The fast file system (ffs) is preferred see fs(4ffs).*

*ff.s51k* reads the i-list and directories of the *special* file, assuming it is a file system. I-node data is saved for files which match the selection criteria. Output consists of the path name for each saved i-node, plus other file information requested using the print *options* below. Output fields are positional. The output is produced in i-node order; fields are separated by tabs. The default line produced by *ff.s51k* is:

path-name i-number

With all *options* enabled, output fields would be:

path-name i-number size uid

The argument *n* in the *option* descriptions that follow is used as a decimal integer (optionally signed), where **+***n* means more than *n*, **−***n* means less than *n*, and *n* means exactly *n*. A day is defined as a 24 hour period.

| | |
|---|---|
| **−I** | Do not print the i-node number after each path name. |
| **−l** | Generate a supplementary list of all path names for multiply-linked files. |
| **−p** *prefix* | The specified *prefix* will be added to each generated path name. The default is . (dot). |
| **−s** | Print the file size, in bytes, after each path name. |
| **−u** | Print the owner's login name after each path name. |
| **−a** *n* | Select if the i-node has been accessed in *n* days. |
| **−m** *n* | Select if the i-node has been modified in *n* days. |
| **−c** *n* | Select if the i-node has been changed in *n* days. |
| **−n** *file* | Select if the i-node has been modified more recently than the argument *file*. |
| **−i** *i-node-list* | Generate names for only those i-nodes specified in *i-node-list*. |

**SEE ALSO**

*ncheck.S51K(1M).*
*find(1)* in the *User's Reference Manual.*

**ERRORS**

If the **−l** option is not specified, only a single path name out of all possible ones is generated for a multiply-linked i-node. If **−l** is specified, all possible names for every linked file on the file system are included in the output. However, no selection criteria apply to the names generated.

**NAME**

fill – fill memory with value

**SYNOPSIS**

**fill** [ **-bhw** ] [ **-v** *value* ] *range*

**DESCRIPTION**

The *fill* command fills the contents of a specified range of addresses with a particular value.

Memory stores occur in either byte, halfword, or word size units. The default width is word (32 bits). An alternative width can be selected by entering one of the following characters as an argument.

**-b** Byte (8 bits)

**-h** Halfword (16 bits)

**-w** Word (32 bits)

The default value used to fill memory is 0, an alternate value may be specified by the **-v** option.

The range specification indicates the amount of memory to be filled. You can specify a range in one of the following ways.

| | |
|---|---|
| base | Fills the contents of the memory address at *base* . |
| base#count | Fills the contents of memory starting at *base* and ending at *base* + *count* . |
| base:limit | Fills the contents of the memory addresses starting at *base* and ending at *limit*. |

**SEE ALSO**

g(1prom), p(1prom), dump(1prom)

**NAME**

finc.s51k – fast incremental backup

**SYNOPSIS**

**/etc/finc.S51K** [ *selection-criteria* ] *file-system raw-tape*

**DESCRIPTION**

**NOTE:** *the obsolete S51K file system has been kept for backward compatibility The fast file system (ffs) is preferred. See fs(4ffs).*

*finc.s51k* selectively copies the input *file-system* to the output *raw-tape* . The cautious will want to mount the input *file-system* read-only to insure an accurate backup, although acceptable results can be obtained in read-write mode. The tape must be previously labelled by *labelit*. The selection is controlled by the *selection-criteria*, accepting only those inodes/files for whom the conditions are true.

It is recommended that production of a *finc.s51k* tape be preceded by the *ff* command, and the output of *ff* be saved as an index of the tape's contents. Files on a *finc.s51k* tape may be recovered with the *frec* command.

The argument *n* in the *selection-criteria* which follow is used as a decimal integer (optionally signed), where **+***n* means more than *n*, **−***n* means less than *n*, and *n* means exactly *n*. A day is defined as a 24 hours.

| | |
|---|---|
| **−a** *n* | True if the file has been accessed in *n* days. |
| **−m** *n* | True if the file has been modified in *n* days. |
| **−c** *n* | True if the i-node has been changed in *n* days. |
| **−n** *file* | True for any file which has been modified more recently than the argument *file*. |

**EXAMPLES**

To write a tape consisting of all files from file-system */usr* modified in the last 48 hours:

```
finc.S51K −m −2 /dev/rdsk/s0d0s6 /dev/rmt/ctape0
```

**SEE ALSO**

*ff.S51K(1M), frec.S51K(1M), labelit(1M).*
*cpio.S51K(1)* in the *User's Reference Manual.*

**NAME**

format –program used for hard disks

**DESCRIPTION**

This manpage describes the standalone program, Format and how it works for SMD disk drives and SCSI disk drives. The Format program is used for formatting hard disks prior to constructing file systems. In addition, the Format program records the bad sectors and constructs an initial volume header. The Format program can also be used to modify a disk partition table or to examine the volume header without formatting the disk.

**CAUTION** SCSI disks are formatted at the factory and do not need to be formatted. The disk format that is performed by the factory is more rigorous and finds more defects than the following Format program is capable of detecting. Therefore, it is recommended that disks are not formatted unless it is believed that there is something physically wrong with the disk.

**DESCRIPTION**

The format program consists of the following phases.

initialize the drive (SMD and SCSI)
read the media defects (SMD only)
format the drive(SMD and SCSI, but see above caution for SCSI)
scan the drive for bad sectors (SMD and SCSI)
manipulate the bad sector list (SMD and SCSI)
map the bad sectors (SMD and SCSI)
write the bad sector list and volume header (SMD and SCSI)
(SCSI writes only the volume header.)

**Phase 1 - Initialize the Drive** During initialization, a valid volume header for the device is obtained, either from the device itself or by constructing one based on user input. **Note:** Contact MIPS customer support before formatting an unsupported drive.

If you are formatting an SMD drive, then the Format program first asks for the name of the device, the controller number, and the unit number. If you are formatting a SCSI drive, then the Format program asks for the name of the device, the LUN number, and the target ID. Based on this information, the program looks for a valid volume header.

If a valid volume header is found and the drive is an SMD, then the program reads the bad sector table from the drive. If the volume header is invalid, then the program provides a list of the known, supported devices. By selecting one of these devices from the displayed list, the program automatically creates a volume header. If your device is not on the list and you select "other", then you are prompted to enter the device parameters. Again, contact MIPS customer support if you wish to format an unsupported drive.

After the drive specific parameters are known, format initializes the partition table. Eleven of the sixteen partitions are initialized. Eight of these partitions are reserved as UNIX partitions, while the other three are marked as the volume directory partition, the track forwarding partition (track forwarding is used for the SMD drive only), and the entire volume partition. The "volume directory partition" contains the bad sector table and the first level boot program. The "track forwarding partition "maps bad tracks (not used for SCSI), and the "entire volume partition" allows access to the entire disk.

Each UNIX partition can be marked as either BSD or System V; generally you should mark the partitions as BSD. The first partition is used as the root and is approximately 16 megabytes. The second is used as the swap and is appoximately 48 megabytes. The third includes all partitions. The seventh is set to the size of all partitions minus the space reserved for the root and swap.

After the program has obtained a valid volume header, a prompt is displayed that asks if you want to modify the device parameters and the partition table information. It is recommended that the device parameters should never be modified. Contact MIPS customer support before changing the device parameters.

The program then allows you to modify the partition table. The partition table can be listed and then entries can be added or deleted. You can also initialize the partition table according to the program's default partitioning scheme, replace a specific partition table entry information, and modify the default bootfile name and partition entry number for the bootfile that are contained in the volume header. The Format program requires a partition to be a multiple number of cylinders, but does not perform any checking on partition overlap.

**Phase 2 - Read the Media Defects (SMD only)** After the volume header information is set, the program asks you whether the media defect information should be read from the drive. Most drives, when shipped directly from the manufacturer and not from MIPS, contain the media defect information on the drive itself. The media defect information cannot be read by the program after the drive is formatted, since the format operation overwrites the media defect information. If instructed to read the defects off the drive, format performs the read, prints each defect as it is encountered, and saves the information.

**Phase 3 - Format the Drive** The program then asks you whether you want to format the drive or not, warning you that this is a destructive operation. If you respond with a yes, the program asks if you want to format the entire drive. The Format program can format a single partition of an SMD disk as well as the entire drive. However, if you are formatting a SCSI drive, you have no control over this and the entire drive is formatted. How you respond to the screen prompt determines what portion of the drive will be formatted, scanned, and mapped. If you only format a singe partition, then the program scans only that partition and maps only the bad sectors within that partition. The program prints a dot on the screen for each cylinder that is formatted to indicate the progress of the formatting phase.

**Phase 4 - Scan the Bad Sectors** This phase of the format program scans for bad sectors. The program scans only the portion of the disk that was previously formatted. The program performs a scan by writing a pattern to the disk and then verifying that this same information can be read without errors. You can specify the number of passes that you want the program to perform. A three-byte pattern is used that rotates on each pass. Performing three passes provides every possible combination for the three-byte pattern. The program lists any bad sectors that are encountered during the scan and saves the information for later use. To indicate the progress of the scanning phase, the program prints a dot on the screen for each cylinder that is scanned.

**Phase 5 - Manipulate the Bad Sector List** The bad sector list manipulation phase allows you to add, delete, initialize, and list the bad sectors that the program is aware of for this drive.

You can add a bad sector to the bad sector list for an SMD drive by entering the cylinder number, the track number (head), the byte position within the track, and the length in bits of the defect. This input is usually added based on the written media defect list received with the drive. These defect lists contain the above mentioned information in both hexadecimal and decimal. The decimal representation, which is usually in parenthesis, should always be used without leading zeros.

You may also add the physical number to the bad sector list for either an SMD drive or a SCSI drive by entering the cylinder, the track number, and the sector number. This input is usually added based on the information provided by the scanning phase.

When adding or deleting an entry, the program verifies each piece of information against the drive's parameters. For invalid input, the program prints an error message to the screen and rejects the information.

**Phase 6 - Map the Bad Sectors** Any bad sectors that exist on the disk should be mapped out to avoid problems once the software is installed. Prior to this phase, the program has gathered the list of bad sectors through some combination of the following: reading the media defects off the drive, scanning for bad sectors, reading the bad sectors off the volume header partition, and gathering media defect information from the user. The program now uses this bad sector list and the features of this device to map out these bad sectors. A sector can be mapped out by slipping a sector, or forwarding an entire track for SMD, or by using the **Reassign Blocks** command for the SCSI drive. The capabilities of this device are included as part of the device specific parameters. Once again, format maps only bad sectors in that portion of the disk that was previously formatted. If none of the disk was formatted, then no bad sectors are mapped. This phase is concluded with an opportunity to print the bad sector table on the screen.

**Phase 7 - Write the Bad Sector List and Volume Header** The final phase of Format consists of allocating space in the volume header partition from the bad sector table and writing the bad sector table and volume header out to the device. If you are formatting a SCSI disk, then only the volume header is written out to device. The volume header is replicated in the first sector of each track of cylinder zero of the device.

## BOOTING THE FORMAT PROGRAM

The standalone version of Format is booted using the PROM Monitor **Boot** command. The Format program can be booted from a cartridge tape, from a hard disk if the software has already been installed from the network. To boot the Format program from the network, a machine must be running the bootfile Server Daemon **bfsd(8)** .

To load the Format program from the cartidge tape containing the release software for SMD drives, type:

```
boot -f tqij (,6,2)format
```

To load the Format program from the cartridge tape containing released software for SCSI drives, type:

```
boot -f tqis()(,,2)format (For an M/120)

or

boot -f tqsd(,6,2)format (For an M/2030)
```

To load the Format program from SMD drive, type:

```
boot dkip()/stand/format
```

To load the Format program from SCSI disk, type:

```
boot dkis() /stand/format (for an M/120)

or

boot dksd()/stand/format (for an M/2030)
```

To load the Format program from the network, type:

```
boot -f bfs()/stand/format
```

The parenthesis in the commands shown above indicate that the previous argument is a device. When booting over the network, it the command is entered as shown, then it will boot

the Format program from the first machine that is found that has the program. You can also boot the format program from a specific machine by specifying the machine name and a path name as shown in the following example.

```
boot -f bfs()machinename:/stand/format
```

After the format program is called using the PROM Monitor **boot** command, several questions are displayed on the screen. The questions are displayed one at a time. Some of the questions require a yes or a no answer, and some of the questions require numeric or typed-word answers. For questions that require a yes or a no answer, the program interprets any character other than *y* to be *no* .

The following pages contain actual screen output from the Format program as it is used to format an SMD drive and a SCSI drive. The questions that are displayed on the screen by the program appear one at a time. In the following examples, the screen output shows related and sequential questions grouped together. In the following examples, the screen output shows related and sequential questions grouped together.

**FORMATTING AN SMD DRIVE**

If you are formatting an SMD drive, then when you first enter the Format program, the following program information and questions are asked.

```
MIPS Format Utility

Version 4.10
date and time appear here

name of device?
controller number?
unit number?
```

If you enter a device name that the program does not recognize, then after you have entered a controller number and a unit number the program displays an error message, lists the known devices, and redisplays the "name of device" question. The device name for SMD drives is dkip.

After you have provided valid input for the first three questions, one of several things happens. First, if the disk has been previously formatted and if there are any bad sectors, then the first message shown below is displayed on the screen. The first message shown below will not appear on the screen if the disk has never been formatted. If the volume header is valid then the second message shown below is displayed on the screen. If the volume header is not valid, or if the disk is new and has never been formatted, then the "device parameters are known for:" screen display shown below is immediately displayed on the screen, and you are asked to enter the number of the device.

```
read in number of defects from 'on disk' bad sector table

choose new drive parameters (y if yes)?

device parameters are known for:
        (0) "fuji 2322 (170Meg unfmtd, 32 sec)"
        (1) "fuji 2333 (337Meg unfmtd, 63 sec)"
        (2) "fuji 2333 (337Meg unfmtddd, 64 sec)"
        (3) "fuji 2344 (689Meg unfmtd, 63 sec)"
        (4) "fuji 2344 (689Meg unfmtd, 64 sec)"
        (5) "fuji 2372 (824Meg unfmtd, 63 sec)"
```

```
(6) "fuji 2372 (824Meg unfmtd, 64 sec)"
(7) "fuji 2372 (824Meg unfmtd, 69 sec)"
(8) "cdc  94161 (156Meg fmtd, SCSI, Wren III)"
(9) "cdc  94171 (328Meg fmtd, SCSI, Wren IV)"
(10) "cdc 94181 (330Meg fmtd, SCSI, Fast Access, Wren V)"
(11) "cdc 94191 (663Meg fmtd, SCSI, Wren VI)"
(12) "cdc 94351 (172Meg fmtd, SCSI, Swift)"
enter number for one of the above?
```

It is recommended that you answer the "choose new drive parameters" question with a no, unless you know that your volume header contains incorrect information. If you answer the question shown above with a yes, then the "device parameters are known for:" screen display, also shown above is displayed on the screen.

After selecting one of the disk drives from the displayed list or answering no to the "choose new drive parameters" question, the following question is displayed on the screen.

```
The UNIX file system partitions may be either ffs(BSD) for System V
do you desire fast(BSD) file system partitions (n if no)?
```

MIPS no longer supports System V file systems. The following question is displayed on the screen.

dump device parameters (y if yes)?

If you answer this question with a yes, then the following information shown below is displayed on the screen. The device parameters for your disk are displayed in place of the *number* in the display shown below:

```
spiral skew = number
number words in gap1 = number
number workds in gap2 = number
number cylinders = number
vol 0 starting head = number
number heads in vol 0 = number
vol 1 starting head = number
number heads in vol1 - number
number sectors per track = number
number bytes per sector = number
sector interleave = number
number retries on error = number
milliseconds per word = number
enabled attributes =
        sector slipping
        track forwarding
        recalibrate as last resort
total bytes per track = number
maximum defect length in bits = number
maximum num defective tracks = number
maximum num defects = number
total bytes per sector = number
modify device parameters (y if yes)?
```

It is recommended that these device parameters should never be modified. If you think you need to change these parameters, then contact MIPS customer support first. When you answer no to the question shown above, the following question is displayed on the screen.

dump partition table (y if yes) ?

If you answer this question with a yes, then the following partition table information is displayed on the screen. The partition table for your disk will be displayed in place of the note below. After the table is displayed, you are asked if you want to modify the partition table. If you had answered no to the "dump partition table" question, then the same question (modify partition table)
would appear on the screen.

Root partition is entry # number
Swap partition is entry # number
Default boot file is /vmunix

Partition Table appears here

modify partition table (y if yes)?

If you choose to modify the partition table, then the following is displayed on the screen.

partition table manipulation
choose one of (list, add, delete, quit, init, modify, replace)
command?

Depending on which item you select, the screen display is different. If the modify table question had been answered with a no, then the following message appears on the screen. This same message appears after you have modified the partition table, and exited the loop by pressing the **Enter** key or entering no (N) answer to both the dump and modify questions.

If the drive is directly from the factory defects can be
read from it ONLY ONCE before it is formatted.
read factor defects from the drive (y if yes)?

If you answer yes to the "read factory defects" question, then the defects are displayed as if you have never formmatted the disk. If the disk has been previously formatted, then an error message is displayed on the screen. After the factory defects have been read, the error message has been displayed, or the question above answered with a no, then the following message is displayed on the screen.

formatting destroys disk data, perform format (y if yes)?

format entire disk (y if yes)?

entry number of partition to format?

The display information for answering the first question shown above with a no answer is given on the following page.

If you answer yes to the "formatting" question shown above, then you are asked if you want to format an entire disk. If you answer no to the second question shown above, then you are asked to enter the number of the partition that you wish to format. The partition number must be between 0 and 15.

If you choose to format all or part of the disk, then the formatting message shown below appears on the screen while the disk is being formatted. While the disk is being formatted, a dot is printed on the screen for each cylinder that is formatted. When the format is complete, the scanning warning shown below appears on the screen.

```
formatting
scanning destroys disk data, perform scan (y if yes)?
```

If you answer yes to the sscanning question, then you are asked how many times you want to scan for bad blocks. Scanning is recommended after formatting because the scanning phase detects any errors on the disk. Only the portion of the disk that was formatted is scanned. It is suggested that you scan three times.

```
number of scans for bad blocks (3 are suggested)

scanning for defects, pass 1 (hit ESCAPE to abort)
scanning for defects, pass 2 .. (hit ESCAPE to abort)
scanning for defects, pass 3 .. (hit ESCAPE to abort)

continues for the number of passes you specified
```

If an error is found while scanning, the following error message is displayed, which indicates the cylinder number and track number of the error. After the error has been recorded, the dot printing is resumed for each cylinder that passes.

```
Error on cyl number, track number
```

If you did not want to format the SMD disk and entered a 'no' answer for the "perform format" question, then the following questions appear on the screen one at a time.

```
formatting wasn't done, perform scan anyway (y if yes)?
scan entire disk (y if yes)?
entry number of partition to scan?
```

If you want to scan all or part of the disk, then answer yes to the first question. If you want to scan the complete disk, then answer yes to the second question. Keep in mind, that scanning destroys disk data. If you do not wish to scan the whole disk, then answer no, and enter the number of the partition you wish to scan when the last message shown above appears on the screen.

If you answered no to the first question shown above, or if you formatted the disk and finished scanning, then the following messages appear on the screen.

```
media defect list manipulation, when prompted,
choose one of (list, add, delete, quit)
command?
```

The **list** operation lists or displays the defect list on the screen. The **add** operation allows you to add a known defect to the list, and the **delete** operation allows you to remove a defect from the list. Selecting **quit** exits the scanning phase, and displays the following question on the screen.

```
mapping destroys disk data, perform map (y if yes)?
```

If you answer yes to this question, then the mapping is performed and the following question is displayed on the screen. The following question is also displayed if you answer no to the question shown above.

dump bad sector table

Answering yes to this question displays the bad sector table. The bad sector table lists the block numbers that slipped. After the table is displayed, a message appears on the screen indicating that the bad sector table is being written to disk. If there are no bad sectors, then there is no table to display and the following question is displayed on the screen. Also, answering no to the "dump bad sector table" question displays the following question.

write new volume header? (y if yes)?

Entering a yes answer writes the volume header to disk and then exits the formatting program. If you enter a no answer, then the Format program is exited, and any changes you made to the device parameters or to the partion table are not saved.

**FORMATTING A SCSI DRIVE**

If you are formatting a SCSI Drive, then when you first enter the Format program, the following program information and questions are asked.

MIPS Format Utility
Version 4.10 Thu June 16 08:42:14 PDT 1988 root

name of device?
LUN number?
target id?

If you enter a device name that the program does not recognize, then after you have entered a LUN number and target id number, the program displays an error message, lists the known devices, and redisplays the "name of device" question. The only valid SCSI disk devices are dkis (M/120), dkij(M/2000) and dksd (M/2030). The only valid SCSI tape devices are tqis (M/120), tqij (M/2000), and tqsd (M/2030).

| | |
|---|---|
| tty: | console uart |
| console: | pseudo console |
| dkis: | SCSI disk |
| bfs: | boot server/LANCE Ethernet |
| tqis: | SCSI tape |
| mem: | memory pseudo-device |

After you have provided valid input for the first three questions, one of two things will happen. First, if the volume header is valid, then the question shown below is diplayed on the screen.

choose new drive parameters (y if yes)?

It is recommended that you answer this question with a no, unless you know that your volume header contains incorrect information. If you answer the question shown above with a yes, then the following information is displayed on the screen.

device parameters are known for:

(0) "fuji 2322 (170Meg unfmtd, 32 sec)"
(1) "fuji 2333 (337Meg unfmtd, 63 sec)"
(2) "fuji 2333 (337Meg unfmtd, 64 sec)"
(3) "fuji 2344 (689Meg unfmtd, 63 sec)"
(4) "fuji 2344 (689Meg unfmtd, 64 sec)"

(5) "fuji 2372 (824Meg unfmtd, 63 sec)"
(6) "fuji 2472 (824Meg unfmtd, 64 sec)"
(7) "fuji 2372 (824Meg unfmtd, 69 sec)"
(8) "cdc 94161 (156Meg fmtd, SCSI, Wren III)"
(9) "cdc 94171 (328Meg fmtd, SCSI, WREN IV)"
(10) "cdc 94181 (330Meg fmtd, SCSI, Fast Access, Wren V)"
(11) "cdc 94191 (663Meg fmtd, SCSI, WREN VI)"
(12) "cdc 94351 (172Meg fmtd, SCSI, Swift)"
enter number for one of the above?

Second, if the volume header is not valid, or if the disk is new and has never been formatted, then the screen display shown above is immediately displayed on the screen, and you are asked to enter the number of the device.

In the display shown above, items 1-8 are not shown because they are for the SMD disk drives which are not used in the M/120 or M/2030. After selecting one of this disk drives from the displayed list, the following question is displayed on the screen.

The UNIX file system partitions may be either ffs (BSD) or
System V. Do you desire fast file system (BSD) partitions?

MIPS no longer supports System V file systems. The following question is displayed on the screen.

dump device parameters(y if yes)?

If you answer this question with a yes, then the following information is displayed on the screen. The device parameters for your disk will be displayed in place of the *number* in the following example.

number cylinders = "number"
number heads = "number"
number sectors pertrack = "number"
number bytes per sector = "number"
sector interleave = "number"
modify device parameters (y if yes)?

If you answer the dump device parameters question with a no (N), then the following question appears on the screen. This is the same question that appears if you had indicated that you wanted to dump the device parameters first.

modify device parameters (y if yes)?

It is recommended that these device parameters should never be modified. If you think you need to change these parameters, then contact MIPS customer support first. When you answer no to the question shown above, the following question is displayed on the screen.

dump partition table (y if yes)?

If you answer this question with a yes, then the following partition table information is displayed on the screen. The partition table for your disk will be displayed in place of the note below. After the table is displayed, you are asked if you want to modify the partition table. If you had answered no to the dump partition table question, then this same question (modify partition table) appears on the screen.

Root partition is entry # "number"
Swap partition is entry # "number"

Default boot file is /vmunix

"Partition Table appears here"

modify partition table (y if yes)?

If you choose to modify the partition table, then the following is displayed on the screen.

partition table manipulation
choose one of (list, add, delete, quit, init, modify, replace)
command?

Depending on which item you select, the screen display is different. If the modify partition table question had been answered with a no, then the following message appears on the screen. This same message appears after you have modified the partition table, and exited the loop by pressing the **Enter** key or entering a no (N) answer to both the dump and modify questions.

formatting destroys ALL SCSI disk data, perform format (y if yes)?

If you answer yes to this question, then the firs message shown below appears on the screen while the disk is being formatted. Formatting takes awhile. When the disk has been formatted, then the second message shown below appears on the screen.

formatting
scanning destroys all disk data, perform scan (y if yes)?

If you answer yes to the above question, then you are asked how many times you want to scan for bad blocks. Scanning is recommended after formatting because it is the scanning phase that detects errors on the disk. It is suggested that you scan three times.

number of scans for bad blocks (3 are suggested)?
starting cylinder is 0, ending cylinder is number

scanning for defects, pass 1 (hit escape to abort)

scanning for defects, pass 2 .. (hit escape to abort)

scanning for defects, pass 3 .. (hit escape to abort)

continues for the number of passes you specified.

If an error is found while scanning, the following error message is displayed, which indicates the cylinder number and track number of the error. After the error has been recorded, the dot printing is resumed for each cylinder that passes.

Error on cyl number, track number

If you did not want to format the SCSI disk and entered a no answer, then the following question appears on the screen.

formatting wasn't done, perform scan anyway (y if yes)?

If you answered no to the question above or if you fromatted the disk and finished the scanning, then the following messages appear on the screen.

```
SCSI defect list manipulation, when prompted,
choose one of (list, add, delete, quit)
command?
```

The **list** operation lists or displays the defect list on the screen. The **add** operation allows you to add a known defect to the list, and the **delete** operation allows you to remove a defect from the list. Selecting **quit** , exits the scanning phase, and displays the following question on the screen.

```
write new volume header? (y if yes)
```

Entering a yes answer writes the volume header to disk and then exits the formatting program. If you enter a no answer, then the Format program is exited, and any changes you made to the device parameters or to the partition table are not saved.

**SEE ALSO**

An example format session is contained in the section entitled **Disk Management Procedures** in the *Systems Administrator's Guide* . Also, additional information on how to create a volume header can be found in the software installation instructions in the *Release Notes* .

**NAME**

frec – recover files from a backup tape

**SYNOPSIS**

**/etc/frec** [ **–p** *path*] [ **–f** *reqfile*] *raw_tape i_number:name* ...

**DESCRIPTION**

*frec* recovers files from the specified *raw_tape* backup tape written by *volcopy(1M)* or *finc(1M)*, given their *i_numbers*. The data for each recovery request will be written into the file given by *name* .

The **–p** option allows you to specify a default prefixing *path* different from your current working directory. This will be prefixed to any *names* that are not fully qualified, i.e. that do not begin with **/** or **./**. If any directories are missing in the paths of recovery *names* they will be created.

**–p** *path* — Specifies a prefixing *path* to be used to fully qualify any names that do not start with **/** or **./**.

**–f** *reqfile* — Specifies a file which contains recovery requests. The format is *i_number:newname,* one per line.

**EXAMPLES**

To recover a file, i_number 1216 when backed-up, into a file named *junk* in your current working directory:

```
frec  /dev/rSA/ctape1  1216:junk
```

To recover files with i_numbers 14156, 1232, and 3141 into files */usr/src/cmd/a*, */usr/src/cmd/b* and */usr/joe/a.c*:

```
frec  -p  /usr/src/cmd  /dev/rSA/ctape1  14156:a  1232:b
  3141:/usr/joe/a.c
```

**SEE ALSO**

*ff(1M), finc(1M), labelit(1M).*
*cpio(1)* in the *User's Reference Manual.*

**ERRORS**

While paving a path (i.e. creating the intermediate directories contained in a pathname) **frec** can only recover inode fields for those directories contained on the tape and requested for recovery.

**NAME**

fsck.ffs – file system consistency check and interactive repair

**SYNOPSIS**

**/etc/fsck.ffs −p** [ *filesystem ...* ]
**/etc/fsck.ffs** [ **−b** *block#* ] [ **−y** ] [ **−n** ] [ *filesystem* ] ...

**DESCRIPTION**

The first form of *fsck.ffs* preens a standard set of filesystems or the specified file systems. It is normally used in the script */etc/rc* during automatic reboot. In this case *fsck.ffs* reads the table */etc/fstab* to determine which file systems to check. It uses the information there to inspect groups of disks in parallel taking maximum advantage of i/o overlap to check the file systems as quickly as possible. Normally, the root file system will be checked on pass 1, other "root" ("a" partition) file systems on pass 2, other small file systems on separate passes (e.g. the "d" file systems on pass 3 and the "e" file systems on pass 4), and finally the large user file systems on the last pass, e.g. pass 5. Only partitions in *fstab* that are mounted *rw* or *rq* and that have non-zero pass number are checked.

The system takes care that only a restricted class of innocuous inconsistencies can happen unless hardware or software failures intervene. These are limited to the following:

Unreferenced inodes

Link counts in inodes too large

Missing blocks in the free list

Blocks in the free list also in files

Counts in the super-block wrong

These are the only inconsistencies that *fsck.ffs* with the **−p** option will correct; if it encounters other inconsistencies, it exits with an abnormal return status and an automatic reboot will then fail. For each corrected inconsistency one or more lines will be printed identifying the file system on which the correction will take place, and the nature of the correction. After successfully correcting a file system, *fsck.ffs* will print the number of files on that file system, the number of used and free blocks, and the percentage of fragmentation.

If sent a QUIT signal, *fsck.ffs* will finish the file system checks, then exit with an abnormal return status that causes the automatic reboot to fail. This is useful when you wish to finish the file system checks, but do not want the machine to come up multiuser.

Without the **−p** option, *fsck.ffs* audits and interactively repairs inconsistent conditions for file systems. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. It should be noted that some of the corrective actions which are not correctable under the **−p** option will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond **yes** or **no**. If the operator does not have write permission on the file system *fsck.ffs* will default to a **−n** action.

*fsck* has more consistency checks than its predecessors *check, dcheck, fcheck,* and *icheck* combined.

The following flags are interpreted by *fsck.ffs.*

**−b** Use the block specified immediately after the flag as the super block for the file system. Block 32 is always an alternate super block.

**−y** Assume a yes response to all questions asked by **fsck.ffs**; this should be used with great caution as this is a free license to continue after essentially unlimited trouble has been encountered.

**-n** Assume a no response to all questions asked by **fsck.ffs**; do not open the file system for writing.

If no filesystems are given to *fsck.ffs* then a default list of file systems is read from the file */etc/fstab*.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Directory size not of proper format.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.
8. Super Block checks:
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the *lost+found* directory. The name assigned is the inode number. If the *lost+found* directory does not exist, it is created. If there is insufficient space its size is increased.

Checking the raw device is almost always faster.

**FILES**

*/etc/fstab* contains default list of file systems to check.

**DIAGNOSTICS**

The diagnostics produced by *fsck.ffs* are fully enumerated and explained in Appendix A of "Fsck – The UNIX File System Check Program" (SMM:5).

**SEE ALSO**

*crash(1M), fsck(1M), fsck(1S51K), mkfs(1FFS), newfs(1FFS) fs(4FFS), fstab(4)* in the *Programmer's Reference Manual.*

**ERRORS**

There should be some way to start a **fsck.ffs -p** at pass *n*.

**WARNING**

*fsck.ffs* reboots itself after executing *fsck* on the root partition if it thinks the reboot is necessary. This prevents possible disk corruption if *fsck.ffs* changes the root file system.

**NAME**

fsck - front-end for filesystem checkers

**SYNOPSIS**

**/etc/fsck** [ *options for specific fsck* ] *filesystem...*

**DESCRIPTION**

This command is a front-end program that collects options and executes the proper *fsck* command for each filesystem. That is, filesystem types may be mixed, and the proper checker will be executed for the filesystem. Currently, only FFS and S51K filesystems are supported.

The options that can be used are described on the respective manual pages for the filesystem-specific *fsck* commands. Note that many options are not common to the two filesystem-specific checkers. Only options understood by a checker is passed on. In addition, the **-b** option syntax is very different for the two commands, so it may not be used.

There is no automatic check list generation, so at least one filesystem must be specified. If this feature is required, the filesystem-specific command needed must be invoked.

**SEE ALSO**

*fsck(1FFS), fsck(1S51K).*

**NAME**

**fsck.s51k** - check and repair file systems

**SYNOPSIS**

**/etc/fsck.s51k** [**−y**] [**−n**] [**−s**X] [**−S**X] [**−t** *file* ] [**−q**] [**−D**] [**−f**] [**−b**] [*file-systems*]

**DESCRIPTION**

**NOTE:** *The obsolete S51K file system has been kept for backward compatibility. The fast file system (FFS) is preferred. See fs(4FFS).*

**Fsck.s51k** audits and interactively repairs inconsistent conditions for file systems. If the file system is found to be consistent, the number of files, blocks used, and blocks free are reported. If the file system is inconsistent the user is prompted for concurrence before each correction is attempted. It should be noted that most corrective actions will result in some loss of data. The amount and severity of data loss may be determined from the diagnostic output. The default action for each correction is to wait for the user to respond **yes** or **no**. If the user does not have write permission **fsck.s51k** defaults to a **−n** action.

The following options are accepted by **fsck.s51k**.

**−y** Assume a **yes** response to all questions asked by **fsck.s51k**.

**−n** Assume a **no** response to all questions asked by **fsck.s51k**; do not open the file system for writing.

**−s***X* Ignore the actual free list and (unconditionally) reconstruct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done; if this is not possible, care should be taken that the system is quiescent and that it is rebooted immediately afterwards. This precaution is necessary so that the old, bad, in-core copy of the superblock will not continue to be used, or written on the file system.

The **−s***X* option allows for creating an optimal free-list organization.

If *X* is not given, the values used when the file system was created are used. The format of *X* is *cylinder size*:*gap size*.

**−S***X* Conditionally reconstruct the free list. This option is like **−s***X* above except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using **−S** will force a **no** response to all questions asked by **fsck.s51k**. This option is useful for forcing free list reorganization on uncontaminated file systems.

**−t** If **fsck.s51k** cannot obtain enough memory to keep its tables, it uses a scratch file. If the **−t** option is specified, the file named in the next argument is used as the scratch file, if needed. Without the **−t** flag, **fsck.s51k** will prompt the user for the name of the scratch file. The file chosen should not be on the file system being checked, and if it is not a special file or did not already exist, it is removed when **fsck.s51k** completes.

**−q** Quiet **fsck.s51k.** Do not print size-check messages. Unreferenced *fifos* will silently be removed. If **fsck.s51k** requires it, counts in the superblock will be automatically fixed and the free list salvaged.

**−D** Directories are checked for bad blocks. Useful after system crashes.

**−f** Fast check. Check block and sizes and check the free list. The free list will be reconstructed if it is necessary.

**−b** Reboot. If the file system being checked is the root file system and modifications have been made, then either remount the root file system or reboot the system. A remount is done only if there was minor damage.

If no *file-systems* are specified, **fsck.s51k** will read a list of default file systems from the file */etc/checklist*.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one i-node or the free list.
2. Blocks claimed by an i-node or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Incorrect number of blocks.
   Directory size not 16-byte aligned.
5. Bad i-node format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated i-node.
   I-node number out of range.
8. Super Block checks:
   More than 65536 i-nodes.
   More blocks for i-nodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free i-node count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the user's concurrence, reconnected by placing them in the *lost+found* directory, if the files are nonempty. The user will be notified if the file or directory is empty or not. Empty files or directories are removed, as long as the **-n** option is not specified. **Fsck.s51k** will force the reconnection of nonempty directories. The name assigned is the i-node number. The only restriction is that the directory *lost+found* must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This is accomplished by making *lost+found*, copying a number of files to the directory, and then removing them (before **fsck.s51k** is executed).

Checking the raw device is almost always faster and should be used with everything but the *root* file system.

**FILES**

*/etc/checklist*  contains default list of file systems to check.

**SEE ALSO**

*fsck(1M), fsck(1FFS), mkfs(1S51K), ncheck(1S51K), crash(1M).*
*uadmin(2), checklist(4), fs(4S51K)* in the *Programmer's Reference Manual.*

**BUGS**

I-node numbers for **.** and **..** in each directory are not checked for validity.

**NAME**

fsdb.s51k – file system debugger

**SYNOPSIS**

**/etc/fsdb.S51K** *special* [ **–** ]

**DESCRIPTION**

**NOTE:** *The obsolete S51K file system has been kept for backward compatibility. The fast file system (ffs) is preferred. See fs(4ffs).*

*fsdb.s51k* can be used to patch up a damaged file system after a crash. It has conversions to translate block and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an i-node. These greatly simplify the process of correcting control block entries or descending the file system tree.

*fsdb.s51k* contains several error-checking routines to verify i-node and block addresses. These can be disabled if necessary by invoking *fsdb.s51k* with the optional **–** argument or by the use of the **O** symbol. (*fsdb.s51k* reads the i-size and f-size entries from the superblock of the file system as the basis for these checks.)

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

*fsdb.s51k* reads a block at a time and will therefore work with raw as well as block I/O. A buffer management routine is used to retain commonly used blocks of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding block.

The symbols recognized by *fsdb.s51k* are:

| | |
|---|---|
| # | absolute address |
| **i** | convert from i-number to i-node address |
| **b** | convert to block address |
| **d** | directory slot offset |
| **+,–** | address arithmetic |
| **q** | quit |
| >,< | save, restore an address |
| **=** | numerical assignment |
| **=+** | incremental assignment |
| **=–** | decremental assignment |
| **" ="** | character string assignment |
| **O** | error checking flip flop |
| **p** | general print facilities |
| **f** | file print facility |
| **B** | byte mode |
| **W** | word mode |
| **D** | double word mode |
| **!** | escape to shell |

The print facilities generate a formatted output in various styles. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the delete character. If a number follows the **p** symbol, that many entries are printed. A check is made to detect block boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current block are printed. The print options available are:

| | |
|---|---|
| **i** | print as i-nodes |
| **d** | print as directories |
| **o** | print as octal words |
| **e** | print as decimal words |
| **c** | print as characters |
| **b** | print as octal bytes |

The **f** symbol is used to print data blocks associated with the current i-node. If followed by a number, that block of the file is printed. (Blocks are numbered from zero.) The desired print option letter follows the block number, if present, or the **f** symbol. This print facility works for small as well as large files. It checks for special devices and that the block pointers used to find the data are not zero.

Dots, tabs, and spaces may be used as function delimiters but are not necessary. A line with just a new-line character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or i-node, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal address followed by the value in octal and decimal. A **.B** or **.D** is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. I-nodes are printed with labeled fields describing each element.

The following mnemonics are used for i-node examination and refer to the current working i-node:

| | |
|---|---|
| **md** | mode |
| **ln** | link count |
| **uid** | user ID number |
| **gid** | group ID number |
| **sz** | file size |
| **a#** | data block numbers (0 – 12) |
| **at** | access time |
| **mt** | modification time |
| **maj** | major device number |
| **min** | minor device number |

**EXAMPLES**

| | |
|---|---|
| 386i | prints i-number 386 in an i-node format. This now becomes the current working i-node. |
| ln=4 | changes the link count for the working i-node to 4. |
| ln=+1 | increments the link count by 1. |

| | |
|---|---|
| fc | prints, in ASCII, block zero of the file associated with the working i-node. |
| 2i.fd | prints the first 32 directory entries for the root i-node of this file system. |
| d5i.fc | changes the current i-node to that associated with the 5th directory entry (numbered from zero) found from the above command. The first logical block of the file is then printed in ASCII. |
| 512B.p0o | prints the superblock of this file system in octal. |
| 2i.a0b.d7=3 | changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line. |
| d7.nm="name" | changes the name field in the directory slot to the given string. Quotes are optional when used with **nm** if the first character is alphabetic. |
| a2b.p0d | prints the third block of the current i-node as directory entries. |

**SEE ALSO**

*fsck.s51k(1M), dir(4S51k), fs(4S51K).*

**NAME**

fsirand.ffs – install random inode generation numbers

**SYNOPSIS**

**fsirand.ffs** [ **–p** ] *special*

**DESCRIPTION**

*fsirand.ffs* installs random inode generation numbers on all the inodes on device *special,* and also installs a filesystem ID in the superblock. This helps increase the security of filesystems exported by NFS.

*fsirand.ffs* must be used only on an unmounted filesystem that has been checked with *fsck(1FFS)*. The only exception is that it can be used on the root filesystem in single-user mode, if the system is immediately re-booted afterwords.

**OPTIONS**

**–p** Print out the generation numbers for all the inodes, but do not change the generation numbers.

**NAME**

fsstat – report file system status

**SYNOPSIS**

**/etc/fsstat** *special_file*

**DESCRIPTION**

*fsstat* reports on the status of the file system on *special_file*. During startup, this command is used to determine if the file system needs checking before it is mounted. *fsstat* succeeds if the file system is unmounted and appears okay. For the root file system, it succeeds if the file system is active and not marked bad.

**SEE ALSO**

*fs(4)* in the *Programmer's Reference Manual*.

**DIAGNOSTICS**

The command has the following exit codes:

0 – the file system is not mounted and appears okay,
(except for root where 0 means mounted and okay).
1 – the file system is not mounted and needs to be checked.
2 – the file system is mounted.
3 – the command failed.

**NAME**

fstyp – determine file system identifier

**SYNOPSIS**

**fstyp** *special*

**DESCRIPTION**

*fstyp* allows the user to determine the file system identifier of mounted or unmounted file systems using heuristic programs. The file system type is required by *mount(2)* and sometimes by *mount(1M)* to mount file systems of different types. The directory */etc/fstyp.d* contains a program for each file system type to be checked; each of these programs applies some appropriate heuristic to determine whether the supplied *special* file is of the type for which it checks. If it is, the program prints on standard output the usual file-system identifier for that type and exits with a return code of 0; otherwise it prints error messages on standard error and exits with a non-zero return code. *fstyp* runs the programs in */etc/fstyp.d* in alphabetical order, passing *special* as an argument; if any program succeeds, its file-system type identifier is printed and **fstyp** exits immediately. If no program succeeds, **fstyp** prints "Unknown_fstyp" to indicate failure.

**WARNING**

The use of heuristics implies that the result of **fstyp** is not guaranteed to be accurate.

**SEE ALSO**

*mount(1M).*
*mount(2), sysfs(2)* in the *Programmer's Reference Manual.*

**NAME**

ftpd - DARPA Internet File Transfer Protocol server

**SYNOPSIS**

**/etc/ftpd** [ **−d** ] [ **−l** ] [ **−t***timeout* ]

**DESCRIPTION**

*ftpd* is the DARPA Internet File Transfer Prototocol server process. The server uses the TCP protocol and listens at the port specified in the "ftp" service specification; see *services(4)*.

If the **−d** option is specified, debugging information is written to the syslog.

If the **−l** option is specified, each ftp session is logged in the syslog.

The ftp server will timeout an inactive session after 15 minutes. If the **−t** option is specified, the inactivity timeout period will be set to *timeout*.

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

| **Request** | **Description** |
|---|---|
| ABOR | abort previous command |
| ACCT | specify account (ignored) |
| ALLO | allocate storage (vacuously) |
| APPE | append to a file |
| CDUP | change to parent of current working directory |
| CWD | change working directory |
| DELE | delete a file |
| HELP | give help information |
| LIST | give list files in a directory ("ls -lg") |
| MKD | make a directory |
| MODE | specify data transfer *mode* |
| NLST | give name list of files in directory ("ls") |
| NOOP | do nothing |
| PASS | specify password |
| PASV | prepare for server-to-server transfer |
| PORT | specify data connection port |
| PWD | print the current working directory |
| QUIT | terminate session |
| RETR | retrieve a file |
| RMD | remove a directory |
| RNFR | specify rename-from file name |
| RNTO | specify rename-to file name |
| STOR | store a file |
| STOU | store a file with a unique name |
| 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 959 are recognized, but not implemented.

The ftp server will abort an active file transfer only when the ABOR command is preceded by a Telnet "Interrupt Process" (IP) signal and a Telnet "Synch" signal in the command Telnet stream, as described in Internet RFC 959.

*ftpd* interprets file names according to the "globbing" conventions used by *csh*(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 */etc/ftpusers*.

3) The user must have a standard shell returned by *getusershell(3)*.

4) If the user name is "anonymous" or "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(4)* and *group(4)* 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.

**NOTES**

*/etc/ftpd* is a symbolic link to */usr/etc/ftpd*.

**ERRORS**

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, reverting to the super-user only when binding addresses to sockets. The possible security holes have been extensively scrutinized, but are possibly incomplete.

**ORIGIN**

4.3 BSD

**NAME**

fuser – identify processes using a file or file structure

**SYNOPSIS**

**/etc/fuser** [ **–ku** ] *files*|*resources* [ **–** ] [ [ **–ku** ] *files*|*resources* ]

**DESCRIPTION**

*fuser* outputs the process IDs of the processes that are using the *files* or remote *resources* specified as arguments. Each process ID is followed by a letter code, interpreted as follows: if the process is using the file as 1) its current directory, the code is **c**, 2) the parent of its current directory (only when the file is being used by the system), the code is **p**, or 3) its root directory, the code is **r**. For block special devices with mounted file systems, all processes using any file on that device are listed. For remote resource names, all processes using any file associated with that remote resource (Remote File Sharing) are reported. (*fuser* cannot use the mount point of the remote resource; it must use the resource name.) For all other types of files (text files, executables, directories, devices, etc.) only the processes using that file are reported.

The following options may be used with *fuser*:

**–u** the user login name, in parentheses, also follows the process ID.

**–k** the SIGKILL signal is sent to each process. Since this option spawns kills for each process, the kill messages may not show up immediately [see *kill(2)*].

If more than one group of files are specified, the options may be respecified for each additional group of files. A lone dash cancels the options currently in force; then, the new set of options applies to the next group of files.

The process IDs are printed as a single line on the standard output, separated by spaces and terminated with a single new line. All other output is written on standard error.

You cannot list processes using a particular file from a remote resource mounted on your machine. You can only use the resource name as an argument.

Any user with permission to read */dev/kmem* and */dev/mem* can use *fuser*. Only the super-user can terminate another user's process.

**FILES**

| | |
|---|---|
| */unix* | for system namelist |
| */dev/kmem* | for system image |
| */dev/mem* | also for system image |

**SEE ALSO**

*mount(1M).*
*ps(1)* in the *User's Reference Manual.*
*kill(2), signal(2)* in the *Programmer's Reference Manual.*

**NAME**

g – get and display contents of memory location

**SYNOPSIS**

**g** [ *-bhw* ] *address*

**DESCRIPTION**

The *get* command is a PROM Monitor and *sash* command, which displays the contents of a single memory location in decimal, hex, and ASCII-character formats.

The *get* command is also a Debug Monitor (dbgmon) command. If you use the *get* command with *dbgmon* , then *address* can also be a register name that displays the contents of the named client register. Client registers can be one of three types of registers: general purpose registers that you can specify as either *r0* through *r31*, or by their compiler-usage names; special purpose registers; and system coprocessor registers of a client. See *dbgmon*(1spp) for more details.

The default memory access width is word. An alternative width can be selected by entering one of the following characters on the command line as an argument.

**-b** Byte (8 bits)

**-h** Halfword (16 bits)

**-w** Word (32 bits)

**SEE ALSO**

p(1prom), dump(1prom), fill(1prom)

**NAME**

getty – set terminal type, modes, speed, and line discipline

**SYNOPSIS**

**/etc/getty** [ **–h** ] [ **–t** *timeout* ] *line* [ *speed* [ *type* [ *linedisc* ] ] ]
**/etc/getty –c** *file*

**DESCRIPTION**

*getty* is a program that is invoked by *init(1M)*. It is the second process in the series, (*init-getty-login-shell*) that ultimately connects a user with the UNIX system. It can only be executed by the super-user; that is, a process with the user-ID of *root*. Initially **getty** prints the login message field for the entry it is using from */etc/gettydefs*. *getty* reads the user's login name and invokes the *login(1)* command with the user's name as argument. While reading the name, *getty* attempts to adapt the system to the speed and type of terminal being used. It does this by using the options and arguments specified.

*Line* is the name of a tty line in */dev* to which *getty* is to attach itself. *Getty* uses this string as the name of a file in the */dev* directory to open for reading and writing. Unless *getty* is invoked with the **–h** flag, *getty* will force a hangup on the line by setting the speed to zero before setting the speed to the default or specified speed. The **–t** flag plus *timeout* (in seconds), specifies that *getty* should exit if the open on the line succeeds and no one types anything in the specified number of seconds.

*Speed*, the optional second argument, is a label to a speed and tty definition in the file */etc/gettydefs*. This definition tells *getty* at what speed to initially run, what the login message should look like, what the initial tty settings are, and what speed to try next should the user indicate that the speed is inappropriate (by typing a <*break*> character). The default *speed* is 300 baud.

*Type*, the optional third argument, is a character string describing to *getty* what type of terminal is connected to the line in question. *getty* recognizes the following types:

| | |
|---|---|
| **none** | default |
| **ds40-1** | Dataspeed40/1 |
| **tektronix,tek** | Tektronix |
| **vt61** | DEC vt61 |
| **vt100** | DEC vt100 |
| **hp45** | Hewlett-Packard 45 |
| **c100** | Concept 100 |

The default terminal is *none*; i.e., any crt or normal terminal unknown to the system. Also, for terminal type to have any meaning, the virtual terminal handlers must be compiled into the operating system. They are available, but not compiled in the default condition.

*Linedisc*, the optional fourth argument, is a character string describing which line discipline to use in communicating with the terminal. Again the hooks for line disciplines are available in the operating system but there is only one presently available, the default line discipline, **LDISC0**.

When given no optional arguments, *getty* sets the *speed* of the interface to 300 baud, specifies that raw mode is to be used (awaken on every character), that echo is to be suppressed, either parity allowed, new-line characters will be converted to carriage return-line feed, and tab expansion performed on the standard output. It types the login message before reading the user's name a character at a time. If a null character (or framing error) is received, it is assumed to be the result of the user pushing the "break" key. This will cause *getty* to attempt the next *speed* in the series. The series that *getty* tries is determined by what it finds in */etc/gettydefs*.

After the user's name has been typed in, it is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *ioctl(2)* ).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, *login* is *exec*'d with the user's name as an argument. Additional arguments may be typed after the login name. These are passed to *login*, which will place them in the environment (see *login(1)* ).

A check option is provided. When *getty* is invoked with the **-c** option and *file*, it scans the file as if it were scanning */etc/gettydefs* and prints out the results to the standard output. If there are any unrecognized modes or improperly constructed entries, it reports these. If the entries are correct, it prints out the values of the various flags. See *ioctl(2)* to interpret the values. Note that some values are added to the flags automatically.

**FILES**

*/etc/gettydefs*
*/etc/issue*

**SEE ALSO**

*ct(1C), init(1M), tty(7).*
*login(1)* in the *User's Reference Manual.*
*ioctl(2), gettydefs(4), inittab(4)* in the *Programmer's Reference Manual.*

**BUGS**

While *getty* understands simple single character quoting conventions, it is not possible to quote certain special control characters used by *getty* . Thus, you cannot login via *getty* and type a #, @, /, !, _, backspace, ^**U**, ^**D**, or & as part of your login name or arguments. *getty* uses them to determine when the end of the line has been reached, which protocol is being used, and what the erase character is. They will always be interpreted as having their special meaning.

**NAME**

go – transfer control

**SYNOPSIS**

**go** [ *entry* ]

**DESCRIPTION**

The *go* command transfers control to code assumed to have been previously loaded with the *boot*(1spp), *load*(1spp), or *sload*(1spp) commands. The *entry* argument is the address of the entry point. If you do not specify *entry* , then the *go* command transfers control to the entry point of the last loaded or booted module.

**BUGS**

When an entry point is not specified, *go* does not check that a module has previously been loaded.

**SEE ALSO**

load(1prom), sload(1prom), boot(1prom)

**NAME**

help – display command syntax

**SYNOPSIS**

**help** [ *commandlist* ]

**DESCRIPTION**

The *help* command displays the syntax for all commands in *commandlist*. The *commandlist* argument can be one or more commands separated with a space. If you do not specify a *commandlist* , then the *help* command displays the syntax for all commands. You can also get help by typing a question mark(?), which also displays the syntax for all commands.

**NAME**

helpadm – make changes to the Help Facility database

**SYNOPSIS**

**/etc/helpadm**

**DESCRIPTION**

The UNIX system Help Facility Administration command, *helpadm,* allows UNIX system administrators and command developers to define the content of the Help Facility database for specific commands and to monitor use of the Help Facility. The *helpadm* command can only be executed by login root, login bin, or a login that is a member of group bin.

The *helpadm* command prints a menu of 3 types of Help Facility data which can be modified, and 2 choices relating to monitoring use of the Help Facility. The five choices are:

- modify *startup* data
- add, modify, or delete a *glossary* term
- add, modify, or delete command data (description, options, examples, and keywords)
- prevent monitoring use of the Help Facility (login root and login bin only)
- permit monitoring use of the Help Facility (login root and login bin only)

The user may make one of the above choices by entering its corresponding letter (given in the menu), or may exit to the shell by typing q (for "quit").

If one of the first three choices is chosen, then the user is prompted for additional information; specifically, which *startup* screen, *glossary* term definition, or command description is to be modified. The user may also be prompted for information to identify whether the changes to the database are additions, modifications, or deletions. If the user is modifying existing data or adding new data, then they are prompted to make the appropriate modifications/additions. If the user is deleting a *glossary* term or a command from the database, then they must respond affirmatively to the next query in order for the deletion to be done. In any case, before the user's changes are final, they must respond affirmatively when asked whether they are sure they want their requested database changes to be done.

By default, *helpadm* will put the user into *ed(1)* to make additions/modifications to database information. If the user wishes to be put into a different editor, then they should set the environment variable **EDITOR** in their environment to the desired editor, and then export **EDITOR .**

If the user chooses to monitor/prevent monitoring use of the Help Facility, the choice made is acted on with no further interaction by the user.

**SEE ALSO**

*ed(1), glossary(1), help(1), locate(1), starter(1), usage(1).*

**WARNINGS**

When the UNIX system is delivered to a customer, */etc/profile* exports the environment variable **LOGNAME.** If */etc/profile* has been changed so that **LOGNAME** is not exported, then the options to monitor/prevent monitoring use of the Help Facility may not work properly.

**FILES**

| | |
|---|---|
| *HELPLOG* | */usr/lib/help/HELPLOG* |
| *helpclean* | */usr/lib/help/helpclean* |

**NAME**

id, whoami – print user and group IDs and names

**SYNOPSIS**

**id** [ **+***format* ]
**whoami**

**DESCRIPTION**

By default, *id* prints a message of the form

**uid=***userid*(*username*) **gid=***groupid*(*groupname*)

If the effective userid is different from the real userid, the "uid" portion of the message is followed by

**euid=***userid*(*username*)

A different effective groupid is handled similarly.

A *format* argument may be given to specify exactly what items should be printed. The format may contain a combination of C-like escape sequences (\n, \r, \f, \b, \f, and \\), %-specifiers (see below), and other characters, and is printed with a following newline.

The available %-specifiers are:

| | |
|---|---|
| %% | % |
| %u | userid number |
| %U | effective userid number |
| %g | groupid number |
| %G | effective groupid number |
| %l | username |
| %L | effective username |
| %n | groupname |
| %N | effective groupname |

The command *whoami* is equivalent to giving the format "+%n", and is provided for BSD compatability.

**SEE ALSO**

*logname(1), getuid(2).*

**NAME**

ifconfig – configure network interface parameters

**SYOPNSIS**

**/etc/ifconfig** *interface address_family* [ *address* [ *dest_address* ] ] [ *parameters* ]
**/etc/ifconfig** *interface* [ *protocol_family* ]

**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 or other operating parameters. The *interface* parameter is a string of the form "name unit", e.g. "en0".

Since an interface may receive transmissions in differing protocols, each of which may require separate naming schemes, it is necessary to specify the *address_family*, which may change the interpretation of the remaining parameters. The address families currently supported are "inet" and "ns".

For the DARPA-Internet family, the address is either a host name present in the host name data base, *hosts(4)*, or a DARPA Internet address expressed in the Internet standard "dot notation". For the Xerox Network Systems(tm) family, addresses are *net:a.b.c.d.e.f*, where *net* is the assigned network number (in decimal), and each of the six bytes of the host number, *a* through *f*, are specified in hexadecimal. The host number may be omitted on 10Mb/s Ethernet interfaces, which use the hardware physical address, and on interfaces other than the first.

The following parameters may be set with **ifconfig**:

**up** Mark an interface "up". This may be used to enable an interface after an "ifconfig down." It happens automatically when setting the first address on an interface. If the interface was reset when previously marked down, the hardware will be re-initialized.

**down** Mark an interface "down". When an interface is marked "down", the system will not attempt to transmit messages through that interface. If possible, the interface will be reset to disable reception as well. This action does not automatically disable routes using the interface.

**trailers** Request 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. On networks that support the Address Resolution Protocol (see *arp(7P)*; currently, only 10 Mb/s Ethernet), this flag indicates that the system should request that other systems use trailers when sending to this host. Similarly, trailer encapsulations will be sent to other hosts that have made such requests. Currently used by Internet protocols only.

**–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.

**metric** *n* Set the routing metric of the interface to *n*, default 0. The routing

metric is used by the routing protocol (*routed(1m)*). Higher metrics have the effect of making a route less favorable; metrics are counted as addition hops to the destination network or host.

**debug** Enable driver dependent debugging code; usually, this turns on extra console error logging.

**−debug** Disable driver dependent debugging code.

**netmask** *mask* (Inet only) Specify how much of the address to reserve for subdividing networks into sub-networks. The mask includes the network part of the local address and the subnet part, which is taken from the host field of the address. The mask can be specified as a single hexadecimal number with a leading 0x, with a dot-notation Internet address, or with a pseudo-network name listed in the network table. The mask contains 1's for the bit positions in the 32-bit address which are to be used for the network and subnet parts, and 0's for the host part. The mask should contain at least the standard network portion, and the subnet field should be contiguous with the network portion.

**dstaddr** Specify the address of the correspondent on the other end of a point to point link.

**broadcast** (Inet only) Specify the address to use to represent broadcasts to the network. The default broadcast address is the address with a host part of all 1's.

**ipdst** (NS only) This is used to specify an Internet host who is willing to receive ip packets encapsulating NS packets bound for a remote network. In this case, an apparent point to point link is constructed, and the address specified will be taken as the NS address and network of the destinee.

*ifconfig* displays the current configuration for a network interface when no optional parameters are supplied. If a protocol family is specified, **ifconfig** will report only the details specific to that protocol family.

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

**DIAGNOSTICS**

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

**SEE ALSO**

*netstat(1), intro(7N), rc(1M)*

**ORIGIN**

4.3 BSD

**NAME**

inetd – internet "super–server"

**SYNOPSIS**

**/etc/inetd** [ **–d** ] [ *configuration file* ]

**DESCRIPTION**

*inetd* should be run at boot time by */etc/rc2.d/S30tcp*. It then listens for connections on certain internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. After the program is finished, it continues to listen on the socket (except in some cases which will be described below). Essentially, *inetd* allows running one daemon to invoke several others, reducing load on the system.

Upon execution, *inetd* reads its configuration information from a configuration file which, by default, is */usr/etc/inetd.conf*. There must be an entry for each field of the configuration file, with entries for each field separated by a tab or a space. Comments are denoted by a "#" at the beginning of a line. There must be an entry for each field. The fields of the configuration file are as follows:

service name or rpc specification
socket type
protocol
wait/nowait
user
server program
server program arguments

The *service name* entry is the name of a valid service in the file */etc/services*. For "internal" services (discussed below), the service name *must* be the official name of the service (that is, the first entry in */etc/services*). Rpc specifications are discussed below.

The *socket type* should be one of "stream", "dgram", "raw", "rdm", or "seqpacket", depending on whether the socket is a stream, datagram, raw, reliably delivered message, or sequenced packet socket.

The *protocol* must be a valid protocol as given in */etc/protocols*. Examples might be "tcp" or "udp".

The *wait/nowait* entry is applicable to datagram sockets only (other sockets should have a "nowait" entry in this space). If a datagram server connects to its peer, freeing the socket so *inetd* can received further messages on the socket, it is said to be a "multi-threaded" server, and should use the "nowait" entry. For datagram servers which process all incoming datagrams on a socket and eventually time out, the server is said to be "single-threaded" and should use a "wait" entry.

"Comsat" ("biff") and "talk" are both examples of the latter type of datagram server. *Tftpd* is an exception; it is a datagram server that establishes pseudo-connections. It must be listed as "wait" in order to avoid a race; the server reads the first packet, creates a new socket, and then forks and exits to allow *inetd* to check for new service requests to spawn new servers.

The *user* entry should contain the user name of the user as whom the server should run. This allows for servers to be given less permission than root. The *server program* entry should contain the pathname of the program which is to be executed by *inetd* when a request is found on its socket. If *inetd* provides this service internally, this entry should be "internal".

The arguments to the server program should be just as they normally are, starting with argv[0], which is the name of the program. If the service is provided internally, the word "internal" should take the place of this entry.

*inetd* provides several "trivial" services internally by use of routines within itself. These services are "echo", "discard", "chargen" (character generator), "daytime" (human readable time), and "time" (machine readable time, in the form of the number of seconds since midnight, January 1, 1900). All of these services are tcp based. For details of these services, consult the appropriate RFC from the Network Information Center.

*inetd* rereads its configuration file when it receives a hangup signal, SIGHUP. Services may be added, deleted or modified when the configuration file is reread.

**NOTE**

*/etc/routed* is symbolic link to */usr/etc/routed*

**SEE ALSO**

*ftpd(1M), rexecd(1M), rlogind(1M), rshd(1M), telnetd(1M), tftpd(1M)*

**ORIGIN**

4.3 BSD

**NAME**

infocmp - compare or print out terminfo descriptions

**SYNOPSIS**

**infocmp** [**-d**] [**-c**] [**-n**] [**-I**] [**-L**] [**-C**] [**-r**] [**-u**] [**-s d|i|l|c**] [**-v**] [**-V**] [**-1**] [**-w** *width*] [**-A** *directory*] [**-B** *directory*] [*termname* ...]

**DESCRIPTION**

*infocmp* can be used to compare a binary *terminfo(4)* entry with other terminfo entries, rewrite a *terminfo(4)* description to take advantage of the **use=** terminfo field, or print out a *terminfo(4)* description from the binary file (*term(4)*) in a variety of formats. In all cases, the boolean fields will be printed first, followed by the numeric fields, followed by the string fields.

**Default Options**

If no options are specified and zero or one *termnames* are specified, the **-I** option will be assumed. If more than one *termname* is specified, the **-d** option will be assumed.

**Comparison Options [-d] [-c] [-n]**

**infocmp** compares the *terminfo(4)* description of the first terminal *termname* with each of the descriptions given by the entries for the other terminal's *termnames*. If a capability is defined for only one of the terminals, the value returned will depend on the type of the capability: **F** for boolean variables, **-1** for integer variables, and **NULL** for string variables.

**-d** produce a list of each capability that is different. In this manner, if one has two entries for the same terminal or similar terminals, using **infocmp** will show what is different between the two entries. This is sometimes necessary when more than one person produces an entry for the same terminal and one wants to see what is different between the two.

**-c** produce a list of each capability that is common between the two entries. Capabilities that are not set are ignored. This option can be used as a quick check to see if the **-u** option is worth using.

**-n** produce a list of each capability that is in neither entry. If no *termnames* are given, the environment variable **TERM** will be used for both of the *termnames*. This can be used as a quick check to see if anything was left out of the description.

**Source Listing Options [-I] [-L] [-C] [-r]**

The **-I**, **-L**, and **-C** options will produce a source listing for each terminal named.

**-I** use the *terminfo(4)* names

**-L** use the long C variable name listed in **<term.h>**

**-C** use the *termcap* names

**-r** when using **-C**, put out all capabilities in *termcap* form If no *termnames* are given, the environment variable **TERM** will be used for the terminal name. The source produced by the **-C** option may be used directly as a *termcap* entry, but not all of the parameterized strings may be changed to the *termcap* format. **infocmp** will attempt to convert most of the parameterized information, but that which it doesn't will be plainly marked in the output and commented out. These should be edited by hand. All padding information for strings will be collected together and placed at the beginning of the string where *termcap* expects it. Mandatory padding (padding information with a trailing '/') will become optional. All *termcap* variables no longer supported by *terminfo(4)*, but which are derivable from other *terminfo(4)* variables, will be output.

Not all *terminfo(4)* capabilities will be translated; only those variables which were part of *termcap* will normally be output. Specifying the **-r** option will take off this restriction, allowing all capabilities to be output in *termcap* form. Note that because padding is collected to the beginning of the capability, not all capabilities are output, mandatory padding is not supported, and *termcap* strings were not as flexible, it is not always possible to convert a *terminfo(4)* string capability into an equivalent *termcap* format. Not all of these strings will be able to be converted. A subsequent conversion of the *termcap* file back into *terminfo(4)* format will not necessarily reproduce the original *terminfo(4)* source. Some common *terminfo* parameter sequences, their *termcap* equivalents, and some terminal types which commonly have such sequences, are:

| Terminfo | Termcap | Representative Terminals |
|---|---|---|
| %p1%c | %. | adm |
| %p1%d | %d | hp, ANSI standard, vt100 |
| %p1%'x'%+%c | %+x | concept |
| %i | %i | ANSI standard, vt100 |
| %p1%?%'x'%>%t%p1%'y'%+%; | %>xy | concept |
| %p2 is printed before %p1 | %r | hp |

**Use= Option [-u]**

**-u** produce a *terminfo(4)* source description of the first terminal *termname* which is relative to the sum of the descriptions given by the entries for the other terminals *termnames*. It does this by analyzing the differences between the first *termname* and the other *termnames* and producing a description with **use=** fields for the other terminals. In this manner, it is possible to retrofit generic terminfo entries into a terminal's description. Or, if two similar terminals exist, but were coded at different times or by different people so that each description is a full description, using **infocmp** will show what can be done to change one description to be relative to the other. A capability will get printed with an at-sign (@) if it no longer exists in the first *termname*, but one of the other *termname* entries contains a value for it. A capability's value gets printed if the value in the first *termname* is not found in any of the other *termname* entries, or if the first of the other *termname* entries that has this capability gives a different value for the capability than that in the first *termname*. The order of the other *termname* entries is significant. Since the terminfo compiler *tic(1M)* does a left-to-right scan of the capabilities, specifying two **use=** entries that contain differing entries for the same capabilities will produce different results depending on the order that the entries are given in. **infocmp** will flag any such inconsistencies between the other *termname* entries as they are found. Alternatively, specifying a capability *after* a **use=** entry that contains that capability will cause the second specification to be ignored. Using **infocmp** to recreate a description can be a useful check to make sure that everything was specified correctly in the original source description. Another error that does not cause incorrect compiled files, but will slow down the compilation time, is specifying extra **use=** fields that are superfluous. **infocmp** will flag any other *termname* **use=** fields that were not needed.

**Other Options** [**−s** d|i|l|c] [**−v**] [**−V**] [**−1**] [**−w** *width*]

| | |
|---|---|
| **−s** | sort the fields within each type according to the argument below: |
| **d** | leave fields in the order that they are stored in the *terminfo* database. |
| **i** | sort by *terminfo* name. |
| **l** | sort by the long C variable name. |
| **c** | sort by the *termcap* name. If no **−s** option is given, the fields printed out will be sorted alphabetically by the *terminfo* name within each type, except in the case of the **−C** or the **−L** options, which cause the sorting to be done by the *termcap* name or the long C variable name, respectively. |
| **−v** | print out tracing information on standard error as the program runs. |
| **−V** | print out the version of the program in use on standard error and exit. |
| **−1** | cause the fields to printed out one to a line. Otherwise, the fields will be printed several to a line to a maximum width of 60 characters. |
| **−w** | change the output to width characters. |

**Changing Databases** [**−A** *directory*] [−B *directory*]

The location of the compiled *terminfo(4)* database is taken from the environment variable **TERMINFO**. If the variable is not defined, or the terminal is not found in that location, the system *terminfo(4)* database, usually in */usr/lib/terminfo*, will be used. The options **−A** and **−B** may be used to override this location. The **−A** option will set **TERMINFO** for the first *termname* and the **−B** option will set **TERMINFO** for the other *termnames*. With this, it is possible to compare descriptions for a terminal with the same name located in two different databases. This is useful for comparing descriptions for the same terminal created by different people. Otherwise the terminals would have to be named differently in the *terminfo(4)* database for a comparison to be made.

## FILES

*/usr/lib/terminfo/?/** compiled terminal description database

## DIAGNOSTICS

malloc is out of space!
: There was not enough memory available to process all the terminal descriptions requested. Run **infocmp** several times, each time including a subset of the desired *termnames*.

use= order dependency found:
: A value specified in one relative terminal specification was different from that in another relative terminal specification.

'use=*term*' did not add anything to the description.
: A relative terminal name did not contribute anything to the final description.

must have at least two terminal names for a comparison to be done.
: The **−u**, **−d** and **−c** options require at least two terminal names.

## SEE ALSO

*tic(1M), curses(3X), term(4), terminfo(4)* in the *Programmer's Reference Manual*.
*captoinfo(1M)* in the *System Administrator's Reference Manual.*
Chapter 10 of the *Programmer's Guide*.

## NOTE

The *termcap* database (from earlier releases of UNIX System V) may not be supplied in future releases.

**NAME**

init, telinit – process control initialization

**SYNOPSIS**

**/etc/init** [ **0123456SsQq** ]

**/etc/telinit** [ **0123456sSQqabc** ]

**DESCRIPTION**

**Init**

*init* is a general process spawner. Its primary role is to create processes from information stored in the file */etc/inittab* (see *inittab(4)*). This file usually has *init* spawn *getty*'s on each line that a user may log in on. It also controls autonomous processes required by any particular system.

*init* considers the system to be in a *run-level* at any given time. A *run-level* can be viewed as a software configuration of the system where each configuration allows only a selected group of processes to exist. The processes spawned by *init* for each of these *run-levels* is defined in the *inittab* file. *init* can be in one of eight *run-levels*, **0–6** and **S** or **s**. The *run-level* is changed by having a privileged user run */etc/init*. This user-spawned *init* sends appropriate signals to the original *init* spawned by the operating system when the system was rebooted, telling it which *run-level* to change to.

*init* is invoked inside the UNIX system as the last step in the boot procedure. First *init* looks in */etc/inittab* for the *initdefault* entry (see *inittab(4)*). If there is one, *init* uses the *run-level* specified in that entry as the initial *run-level* to enter. If this entry is not in */etc/inittab*, *init* requests that the user enter a *run-level* from the virtual system console, */dev/console*. If an **S** or an **s** is entered, *init* goes into the SINGLE USER state. This is the only *run-level* that doesn't require the existence of a properly formatted */etc/inittab* file. If it doesn't exist, then by default the only legal *run-level* that *init* can enter is the SINGLE USER state. In the SINGLE USER state the virtual console terminal */dev/console* is opened for reading and writing and the command */bin/su* is invoked immediately. To exit from the SINGLE USER state, use either *init* or **telinit**, to signal *init* to change the *run-level* of the system. Note that if the shell is terminated (via an end-of-file), *init* will only re-initialize to the SINGLE USER state.

When attempting to boot the system, failure of *init* to prompt for a new *run-level* may be due to the fact that the device */dev/console* is linked to a device other than the physical system console (*/dev/contty*). If this occurs, *init* can be forced to relink */dev/console* by typing a delete on the system console which is colocated with the processor.

When *init* prompts for the new *run-level*, the operator may enter only one of the digits **0** through **6** or the letters **S** or **s**. If **S** or **s** is entered, *init* operates as previously described in the SINGLE USER state with the additional result that */dev/console* is linked to the user's terminal line, thus making it the virtual system console. A message is generated on the physical console, */dev/contty*, saying where the virtual terminal has been relocated.

When *init* comes up initially and whenever it switches out of SINGLE USER state to normal run states, it sets the *ioctl(2)* states of the virtual console, */dev/console*, to those modes saved in the file */etc/ioctl.syscon*. This file is written by *init* whenever the SINGLE USER state is entered.

If a **0** through **6** is entered *init* enters the corresponding *run-level*. Any other input will be rejected and the user will be re-prompted. Note that, on the 3B2 Computer, the *run-levels* **0**, **1**, **5**, and **6** are reserved states for shutting the system down; the *run-levels* **2**, **3**, and **4** are available as normal operating states.

If this is the first time *init* has entered a *run-level* other than SINGLE USER, *init* first scans *inittab* for special entries of the type *boot* and *bootwait*. These entries are performed, providing the *run-level* entered matches that of the entry before any normal processing of *inittab* takes

place. In this way any special initialization of the operating system, such as mounting file systems, can take place before users are allowed onto the system. The *inittab* file is scanned to find all entries that are to be processed for that *run-level*.

*run-level* **2** is defined to contain all of the terminal processes and daemons that are spawned in the multi-user environment. Hence, it is commonly referred to as the MULTI-USER state. *run-level* **3** is defined to start up remote file sharing processes and daemons as well as mount and advertise remote resources. So, *run-level* **3** extends multi-user mode and is know as the Remote File Sharing state. *run-level* **4** is available to be defined as an alternative multi-user environment configuration, however, it is not necessary for system operation and is usually unused.

In a MULTI-USER environment, the *inittab* file is set up so that *init* will create a process for each terminal on the system that the administrator sets up to respawn.

For terminal processes, ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as the result of hanging up. When *init* receives a signal telling it that a process it spawned has died, it records the fact and the reason it died in */etc/utmp* and */etc/wtmp* if it exists (see *who(1)*). A history of the processes spawned is kept in */etc/wtmp*.

To spawn each process in the *inittab* file, *init* reads each entry and for each entry that should be respawned, it forks a child process. After it has spawned all of the processes specified by the *inittab* file, *init* waits for one of its descendant processes to die, a powerfail signal, or until *init* is signaled by *init* or **telinit** to change the system's *run-level*. When one of these conditions occurs, *init* re-examines the *inittab* file. New entries can be added to the *inittab* file at any time; however, *init* still waits for one of the above three conditions to occur. To get around this, **init Q** or **init q** command wakes *init* to re-examine the *inittab* file immediately.

If *init* receives a *powerfail* signal (*SIGPWR*) it scans *inittab* for special entries of the type *powerfail* and *powerwait*. These entries are invoked (if the *run-levels* permit) before any further processing takes place. In this way *init* can perform various cleanup and recording functions during the powerdown of the operating system. Note that in the SINGLE-USER state only *powerfail* and *powerwait* entries are executed.

When *init* is requested to change *run-levels* (via *telinit*), *init* sends the warning signal (**SIGTERM**) to all processes that are undefined in the target *run-level*. *init* waits 5 seconds before forcibly terminating these processes via the kill signal (**SIGKILL**).

**Telinit**

**Telinit**, which is linked to */etc/init*, is used to direct the actions of **init**. It takes a one-character argument and signals *init* via the *kill* system call to perform the appropriate action. The following arguments serve as directives to *init*.

| | |
|---|---|
| **0–6** | tells *init* to place the system in one of the *run-levels* **0–6**. |
| **a,b,c** | tells *init* to process only those */etc/inittab* file entries having the **a**, **b** or **c** *run-level* set. These are pseudo-states, which may be defined to run certain commands, but which do not cause the current *run-level* to change. |
| **Q,q** | tells *init* to re-examine the */etc/inittab* file. |
| **s,S** | tells *init* to enter the single user environment. When this level change is effected, the virtual system teletype, */dev/console*, is changed to the terminal from which the command was executed. |

**FILES**

*/etc/inittab*
*/etc/utmp*
*/etc/wtmp*
*/etc/ioctl.syscon*

*/dev/console*
*/dev/contty*

**SEE ALSO**

*getty(1M), termio(7).*
*login(1), sh(1), who(1)* in the *User's Reference Manual.*
kill(2), inittab(4), utmp(4) in the *Programmer's Reference Manual.*

**DIAGNOSTICS**

If *init* finds that it is respawning an entry from */etc/inittab* more than 10 times in 2 minutes, it will assume that there is an error in the command string in the entry, and generate an error message on the system console. It will then refuse to respawn this entry until either 5 minutes has elapsed or it receives a signal from a user-spawned *init* (**telinit**). This prevents *init* from eating up system resources when someone makes a typographical error in the *inittab* file or a program is removed that is referenced in the *inittab.*

**WARNINGS**

**Telinit** can be run only by someone who is super-user or a member of group *sys.*

If the key switch on an M120 system is in the **LOCKED** position, the state of the system can not be changed. A diagnostic is printed to this effect.

**ERRORS**

Attempting to relink */dev/console* with */dev/contty* by typing a delete on the system console does not work.

**NAME**

init – initialize prom monitor

**SYNOPSIS**

**init**

**DESCRIPTION**

The *init* command reinitializes the PROM Monitor software state; however, the environment variables stored in non-volatile RAM are preserved.

**init_tod**

init_tod-initializes time-of-day chip

SYNOPSIS

**init_tod** [*secs*]

DESCRIPTION

The *init_tod* command initializes the time-of-day chip. It is very important that the time-of-day chip is running; otherwise, the operating system will not work properly. This command is normally executed at the factory.

The argument *secs* is the number of seconds since 1972. To use this command, type *init_tod* without any arguments, then run the *date (1)* command after the operating system has booted.

SEE ALSO

*pr_tod(1prom), data(1)*

**NAME**

install – install commands

**SYNOPSIS**

**/etc/install** [**−c** *dira*] [**−f** *dirb*] [**−i**] [**−n** *dirc*] [**−m** *mode*] [**−u** *user*] [**−g** *group*] [**−o**] [**−s**] *file* [*dirx* . . .]

**DESCRIPTION**

The **install** command is most commonly used in "makefiles" [See *make(1)*] to install a *file* (updated target file) in a specific place within a file system. Each *file* is installed by copying it into the appropriate directory, thereby retaining the mode and owner of the original command. The program prints messages telling the user exactly what files it is replacing or creating and where they are going.

If no options or directories (*dirx* . . .) are given, **install** will search a set of default directories (*/bin*, */usr/bin*, */etc*, */lib*, and */usr/lib*, in that order) for a file with the same name as *file*. When the first occurrence is found, **install** issues a message saying that it is overwriting that file with *file*, and proceeds to do so. If the file is not found, the program states this and exits without further action.

If one or more directories (*dirx* . . .) are specified after *file*, those directories will be searched before the directories specified in the default list.

The meanings of the options are:

**−c** *dira* — Installs a new command (*file*) in the directory specified by *dira*, only if it is not found. If it is found, **install** issues a message saying that the file already exists, and exits without overwriting it. May be used alone or with the **−s** option.

**−f** *dirb* — Forces *file* to be installed in given directory, whether or not one already exists. If the file being installed does not already exist, the mode and owner of the new file will be set to *755* and *bin*, respectively. If the file already exists, the mode and owner will be that of the already existing file. May be used alone or with the **−o** or **−s** options.

**−i** — Ignores default directory list, searching only through the given directories (*dirx* . . .). May be used alone or with any other options except **−c** and **−f**.

**−n** *dirc* — If *file* is not found in any of the searched directories, it is put in the directory specified in *dirc*. The mode and owner of the new file will be set to *755* and *bin*, respectively. May be used alone or with any other options except **−c** and **−f**.

**−m** *mode* — The mode of the new file is set to *mode*. Only available to the superuser.

**−u** *user* — The owner of the new file is set to *user*. Only available to the superuser.

**−g** *group* — The group id of the new file is set to *group*. Only available to the superuser.

**−o** — If *file* is found, this option saves the "found" file by copying it to **OLD***file* in the directory in which it was found. This option is useful when installing a frequently used file such as */bin/sh* or */etc/getty*, where the existing file cannot be removed. May be used alone or with any other options except **−c**.

**−s** — Suppresses printing of messages other than error messages. May be

used alone or with any other options.

**SEE ALSO**

*make(1).*

**NAME**

intro – introduction to maintenance commands and application programs

**DESCRIPTION**

This section describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes. The commands in this section should be used along with those listed in Section 1 of the *User's Reference Manual* and Sections 1, 2, 3, 4, and 5 of the *Programmer's Reference Manual*. References of the form *name(1), (2), (3), (4)* and *(5)* refer to entries in the above manuals. References of the form *name(1M)* or *name(7)* refer to entries in this manual.

**COMMAND SYNTAX**

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

*name* [*option*(*s*)] [*cmdarg*(*s*)]
where:

| | |
|---|---|
| *name* | The name of an executable file. |
| *option* | – *noargletter*(*s*) or,<br>– *argletter*<>*optarg*<br>where <> is optional white space. |
| *noargletter* | A single letter representing an option without an argument. |
| *argletter* | A single letter representing an option requiring an argument. |
| *optarg* | Argument (character string) satisfying preceding *argletter*. |
| *cmdarg* | Path name (or other command argument) *not* beginning with – or, – by itself indicating the standard input. |

**SEE ALSO**

*getopt(1)* in the *User's Reference Manual*.
*getopt(3C)* in the *Programmer's Reference Manual*.

**DIAGNOSTICS**

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program (see *wait(2)* and *exit(2)*). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously "exit code", "exit status", or "return code", and is described only where special conventions are involved.

**ERRORS**

Regrettably, not all commands adhere to the aforementioned syntax.

**NAME**

killall – kill all active processes

**SYNOPSIS**

**/etc/killall** [ *signal* ]

**DESCRIPTION**

*killall* is used by */etc/shutdown* to kill all active processes not directly related to the shutdown procedure.

*killall* terminates all processes with open files so that the mounted file systems will be unbusied and can be unmounted.

*killall* sends *signal* (see *kill(1)*) to all processes not belonging to the above group of exclusions. If no *signal* is specified, a default of **9** is used.

**FILES**

*/etc/shutdown*

**SEE ALSO**

*fuser(1M), shutdown(1M).*
*kill(1), ps(1)* in the *User's Reference Manual.*
*signal(2)* in the *Programmer's Reference Manual.*

**WARNINGS**

The **killall** command can be run only by the super-user.

**NAME**

labelit.s51k – provide labels for file systems

**SYNOPSIS**

**/etc/labelit.s51k** *special* [ *fsname volume* [ **–n** ] ]

**DESCRIPTION**

**NOTE:** *The obsolete S51K file system has been kept for backward compability. The fast file system (ffs) is preferred. See fs(4ffs).*

*labelit.s51k* can be used to provide labels for unmounted disk file systems or file systems being copied to tape. The **–n** option provides for initial labeling only (this destroys previous contents).

With the optional arguments omitted, **labelit.s51k** prints current label values.

The *special* name should be the physical disk section (e.g., */dev/dsk/ips0d0s6*), or the cartridge tape (e.g., */dev/mt/ctape0*). The device may not be on a remote machine.

The *fsname* argument represents the mounted name (e.g., *root*, *u1*, etc.) of the file system.

*volume* may be used to equate an internal name to a volume name applied externally to the disk pack, diskette or tape.

For file systems on disk, *fsname* and *volume* are recorded in the superblock.

**SEE ALSO**

*sh(1)* in the *User's Reference Manual*.
*fs.s51k(4)* in the *Programmer's Reference Manual*.

**NAME**

lboot, mboot – configure bootable kernel

**SYNOPSIS**

**/etc/lboot** [**–v**] [**–m** *master* ] [**–s** *system* ] [**–b** *directory* ] [**–u** *unix*]
**/etc/mboot** [**–v**] [**–m** *master* ] [**–s** *system*[*.suffix*] ] [**–b** *directory* ] [**–u** *unix*]

**DESCRIPTION**

The *lboot* command is used to configure a bootable UNIX kernel. Master files in the directory *master* contain configuration information used by *lboot* when creating a kernel. The file *system* is used by *lboot* to determine which modules are to be configured into the kernel.

The *mboot* command is used to help configure a bootable UNIX kernel. Master files in the directory *master* contain configuration information which is used to create *master[.suffix].c*. *mboot* also creates a file called *objlist[.suffix]* which contains a list of the objects needed to be linked into the kernel. When the file *master[.suffix].c* is compiled, it can then be linked with *kernel.o* and all the objects listed in *objlist[.suffix]* the achieve a fully resolved and bootable UNIX kernel.

If a module in *master* is specified in the *system* file via "INCLUDE:", that module will be included in the bootable kernel. For all included modules, **lboot** searches the *boot* directory for an object file with the same name as the file in *master*, but with a ".o" or ".a" appended. If found, this object is included when building the bootable kernel.

For every module in the *system* file specified via "VECTOR:", *lboot* takes actions to determine if a hardware device corresponding to the specified module exists. Generally, the action is a memory read at a specified base, of the specfied size. If the read succeeds, the device is assumed to exist, and its module will also be included in the bootable kernel.

To create the new bootable object file, the applicable master files are read and the configuration information is extracted and compiled. The output of this compilation is then linked with all included object files.

Master files that are specified in the *system* file via "EXCLUDE:" are also examined; stubs are created for routines specified in the excluded master files that are not found in the included objects.

The options are:

**-m** *master* — This option specifies the directory containing the master files to be used for the bootable kernel. The default *master* directory is *$ROOT/usr/sysgen/master.d*.

**-s** *system* — This option specifies the name of the system file. The default *system* file is *$ROOT/usr/sysgen/system*.

**-b** *directory* — This option specifies the directory where object files are to be found. The default output *directory* is *$ROOT/usr/sysgen/boot*.

**-v** — This option makes *lboot* slightly more verbose.

**-u** *unix* — This option specifies the name of the target kernel. By default, it is *unix.new*, unless the **–t** option is used, in which case the default is *unix.install*.

**-d** — This option displays debugging information about the devices and modules put in the kernel.

**-t** — This option tests if the existing kernel is up-to-date. If the kernel is not up-to-date, it prompts you to proceed. It compares the modification dates of the *system* file, the object files in the *boot* directory, and the configuration files in the *master* directory with that of the output kernel.

It also "probes" for the devices specified with "VECTOR:" lines in the system file. If the devices have been added or removed, or if the kernel is out-of-date, it builds a new kernel, adding ".install" to the target name.

**EXAMPLE**

lboot -s newsystem

This will read the file named *newsystem* to determine which objects should be configured into the bootable object.

**SEE ALSO**

*master(4), system(4)*

**NAME**

link, unlink – link and unlink files and directories

**SYNOPSIS**

**/etc/link** *file1 file2*
**/etc/unlink** *file*

**DESCRIPTION**

The *link* command is used to create a file name that points to another file. Linked files and directories can be removed by the *unlink* command; however, it is strongly recommended that the *rm(1)* and rmdir(1) commands be used instead of the *unlink* command.

The only difference between *ln(1)* and *link/unlink* is that the latter do exactly what they are told to do, abandoning all error checking. This is because they directly invoke the *link(2)* and *unlink(2)* system calls.

**SEE ALSO**

*rm(1)* in the *User's Reference Manual.*
*link(2), unlink(2)* in the *Programmer's Reference Manual.*

**WARNINGS**

These commands can be run only by the super-user.

NAME

load – download image via serial line

SYNOPSIS

**load** console_device

DESCRIPTION

*load* allows you to load memory over a serial line connection from a system running the UMIPS program *tip (1)* . To download an image, use the *tip* command to establish communication with either the local or remote console port of the machine to be downloaded. For additional information, refer to the *tip* command.

If you transfer data to the remote port, be sure that the remote port is enabled. Refer to the *enable* command for additional information. After trying several PROM Monitor commands to verify that *tip (1)* is communicating successfully with the remote port, enter the *load* command, specifying either tty(0) or tty(1) to reflect the serial port with which *tip* is communicating. After the *load* command, the PROM expects you to download an image. If you want to abort this mode, type a Control-c. To download the image, refer to the *tip* command in the User's Reference Manual. The PROM Monitor returns to command mode after the download completes. Use the PROM Monitor *go* command to run the downloaded program.

SEE ALSO

sdownload(1spp), enable(1prom), go(1prom), intro(1spp), sload(1prom), slp(5spp), tip(1spp)

BUGS

*tip (1)* is not yet supported.

**NAME**

lpadmin – configure the LP spooling system

**SYNOPSIS**

**/usr/lib/lpadmin −p** *printer* [*options*]
**/usr/lib/lpadmin −x** *dest*
**/usr/lib/lpadmin −d**[*dest*]

**DESCRIPTION**

*lpadmin* configures line printer (LP) spooling systems to describe printers, classes and devices. It is used to add and remove destinations, change membership in classes, change devices for printers, change printer interface programs and to change the system default destination. *lpadmin* may not be used when the LP scheduler, *lpsched(1M)*, is running, except where noted below.

Exactly one of the **−p**, **−d** or **−x** options must be present for every legal invocation of **lpadmin**.

**−p***printer* names a *printer* to which all of the *options* below refer. If *printer* does not exist then it will be created.

**−x***dest* removes destination *dest* from the LP system. If *dest* is a printer and is the only member of a class, then the class will be deleted, too. No other *options* are allowed with **−x**.

**−d**[*dest*] makes *dest*, an existing destination, the new system default destination. If *dest* is not supplied, then there is no system default destination. This option may be used when *lpsched(1M)* is running. No other *options* are allowed with **−d**.

The following *options* are only useful with **−p** and may appear in any order. For ease of discussion, the printer will be referred to as *P* below.

**−c***class* inserts printer *P* into the specified *class*. *class* will be created if it does not already exist.

**−e***printer* copies an existing *printer's* interface program to be the new interface program for *P*.

**−h** indicates that the device associated with *P* is hardwired. This *option* is assumed when adding a new printer unless the **−l** *option* is supplied.

**−i***interface* establishes a new interface program for *P*. *Interface* is the path name of the new program.

**−l** indicates that the device associated with *P* is a login terminal. The LP scheduler, *lpsched*, disables all login terminals automatically each time it is started. Before re-enabling *P*, its current *device* should be established using *lpadmin* .

**−m***model* selects a model interface program for *P*. *model* is one of the model interface names supplied with the LP Spooling Utilities (see *models* below).

**−r***class* removes printer *P* from the specified *class*. If *P* is the last member of the *class*, then the *class* will be removed.

**−v***device* associates a new *device* with printer *P*. *device* is the pathname of a file that is writable by *lp*. Note that the same *device* can be associated with more than one *printer*. If only the **−p** and **−v** *options* are supplied, then *lpadmin* may be used while the scheduler is running.

**Restrictions.**

When creating a new printer, the **−v** option and one of the **−e**, **−i** or **−m** options must be supplied. Only one of the **−e**, **−i** or **−m** options may be supplied. The **−h** and **−l** keyletters

are mutually exclusive. Printer and class names may be no longer than 14 characters and must consist entirely of the characters **A-Z**, **a-z**, **0-9** and _ (underscore).

**Models.**
Model printer interface programs are supplied with the LP Spooling Utilities. They are shell procedures which interface between *lpsched* and devices. All models reside in the directory */usr/spool/lp/model* and may be used as is with *lpadmin* **−m**. Copies of model interface programs may also be modified and then associated with printers using *lpadmin* **−i**. The following describes the *models* which may be given on the *lp* command line using the **−o** keyletter:

LQP-40 Letter quality printer using XON/XOFF protocol at 9600 baud.

DQP-10 Dot matrix draft quality printer using XON/XOFF protocol at 9600 baud.

**EXAMPLES**

1. For a DQP-10 printer named cI8, it will use the DQP-10 model interface after the command:

   /usr/lib/lpadmin −pcI8 −mdqp10

2. A LQP-40 printer called pr1 can be added to the lp configuration with the command:

   /usr/lib/lpadmin −ppr1 −v/dev/contty −mlqp40

**FILES**

*/usr/spool/lp/**

**SEE ALSO**

*accept(1M), lpsched(1M).*
*enable(1), lp(1), lpstat(1)* in the *User's Reference Manual.*

**NAME**

lpsched, lpshut, lpmove – start/stop the LP scheduler and move requests

**SYNOPSIS**

**/usr/lib/lpsched**
**/usr/lib/lpshut**
**/usr/lib/lpmove** *requests dest*
**/usr/lib/lpmove** *dest1 dest2*

**DESCRIPTION**

*lpsched* schedules requests taken by *lp(1)* for printing on line printers (LP's).

*lpshut* shuts down the line printer scheduler. All printers that are printing at the time **lpshut** is invoked will stop printing. Requests that were printing at the time a printer was shut down will be reprinted in their entirety after **lpsched** is started again.

**Lpmove** moves requests that were queued by *lp(1)* between LP destinations. This command may be used only when *lpsched* is not running.

The first form of the command moves the named *requests* to the LP destination, *dest*. *Requests* are request ids as returned by **lp(1)**. The second form moves all requests for destination *dest1* to destination *dest2*. As a side effect, *lp(1)* will reject requests for *dest1*.

Note that **lpmove** never checks the acceptance status (see *accept(1M)*) for the new destination when moving requests.

**FILES**

*/usr/spool/lp/**

**SEE ALSO**

*accept(1M), lpadmin(1M).*
*enable(1), lp(1), lpstat(1)* in the *User's Reference Manual.*

**NAME**

makedbm – make a dbm file

**SYNOPSIS**

**makedbm** [ **–i** *yp_input_file* ] [ **–o** *yp_output_name* ]
[ **–d** *yp_domain_name* ] [ **–m** *yp_master_name* ]
*infile outfile* **makedbm** [ **–u** *dbmfilename* ]

**DESCRIPTION**

*makedbm* takes *infile* and converts it to a pair of files in *dbm* format, namely *outfile*.pag and *outfile*.dir. Each line of the input file is converted to a single *dbm* record. All characters up to the first tab or space form the key, and the rest of the line is the data. If a line ends with \, then the data for that record is continued on to the next line. *Infile* can be –, in which case standard input is read.

*makedbm* is meant to be used in generating *dbm* files and it generates a special entry with the key *yp_last_modified*, which is the date of *infile* (or the current time, if *infile* is –).

**OPTIONS**

| | |
|---|---|
| **–i** | Create a special entry with the key *yp_input_file*. |
| **–o** | Create a special entry with the key *yp_output_name*. |
| **–d** | Create a special entry with the key *yp_domain_name*. |
| **–m** | Create a special entry with the key *yp_master_name*. If no master host name is specified, *yp_master_name* will be set to the local host name. |
| **–u** | Undo a *dbm* file. That is, print out a *dbm* file one entry per line, with a single space separating keys from values. |

**EXAMPLE**

It is easy to write shell scripts to convert standard files such as */etc/passwd* to the key value form used by *makedbm*. For example,

```
#!/bin/awk -f
BEGIN { FS = ":"; OFS = "\t"; }
{ print $1, $0 }
```

takes the */etc/passwd* file and converts it to a form that can be read by **makedbm** to make the file *passwd.byname*. That is, the key is a username, and the value is the remaining line in the */etc/passwd* file.

**ORIGIN**

Sun Microsystems

**NAME**

mipsinstall – install system files

**SYNOPSIS**

**mipsinstall** [ **–l** *link* ] [ **–L** *link* ] [ **–c** ] [ **–m** *mode* ] [ **–o** *owner* ] [ **–g** *group* ] [ **–s** ] [ **–f** ] *file destination*

**DESCRIPTION**

The named *file* is moved (or copied if **–c** is specified) to *destination*. If the destination is a directory then *file* is moved into the *destination* directory with its original file-name.

Unless the **–f** option is given, the *destination* is taken to be a directory, and is created if it doesn't exist.

If the destination file already exists, it is removed. If it can not be removed, it is renamed. The new name is the old name prefixed with a pound character (#) and suffixed with *.yymmddhhmmss*, where *yy* is the year, *mm* is the month, and so forth. The prefix is used to mark the files for automatic cleanup, and the suffix is used to avoid clashes in case of multiple installations.

The mode for *destination* is set to 755; the **–m** *mode* option may be used to specify a different mode.

*Destination* is changed to owner root; the **–o** *owner* option may be used to specify a different owner.

*Destination* is changed to group staff; the **–g** *group* option may be used to specify a different group.

If the **–s** option is specified the file is stripped after being installed.

If the **–l** option is specified, the "filename" is created as a symbolic link to the installed file. Multiple **–l** options are allowed.

If the **–L** option is specified, the "filename" is created as a hard link to the installed file. Multiple **–L** options are allowed. Hard links may not be made across filesystems.

**Mipsinstall** refuses to move a file onto itself.

**SEE ALSO**

*chgrp(1), chmod(1), cp(1), ln(1), mv(1), strip(1),*

**NAME**

mkboottape – make a boot tape

**SYNOPSIS**

**mkboottape** [**-f** *output_file_name* ] *file1 file2 ... file20*

**DESCRIPTION**

*mkboottape* builds a special directory which contains the list of file names and their logical block number relative to the beginning of the output file. There may be no more than twenty files and the file names must be less than or equal to sixteen characters. The **-f** option may be used to specify an alternate output file or device. Typical use for this program is to create the output file which is *dd*'ed to tape using a blocking factor of 16K. The prom monitor knows about this special directory and can boot any one of the files found in the directory. Unless you have means to create a stand alone program this utility is useless.

**AUTHOR**

Rick McNeal

**NAME**

mkfs.ffs – construct a file system

**SYNOPSIS**

**/etc/mkfs.ffs** [*–N*] *special size* [ *nsect* [ *ntrack* [ *blksize* [ *fragsize* [ *ncpg* [ *minfree* [ *rps* [ *nbpi* [ *opt* ] ] ] ] ] ] ] ] ]

**DESCRIPTION**

**N.B.:** file system are normally created with the *newfs(1FFS)* command.

*mkfs.ffs* constructs a file system by writing on the special file *special* unless the *-N* flag has been specified. The numeric size specifies the number of sectors in the file system. *mkfs.ffs* builds a file system with a root directory and a *lost+found* directory. (See *fsck(1FFS)*.) The number of i-nodes is calculated as a function of the file system size. No boot program is initialized by **mkfs.ffs** (see *newfs(1FFS)*).

The optional arguments allow fine tune control over the parameters of the file system. *nsect* specifies the number of sectors per track on the disk. *ntrack* specifies the number of tracks per cylinder on the disk. *blksize* gives the primary block size for files on the file system. It must be a power of two, currently selected from 4096 or 8192. *fragsize* gives the fragment size for files on the file system. The *fragsize* represents the smallest amount of disk space that will be allocated to a file. It must be a power of two currently selected from the range 512 to 8192. *ncpg* specifies the number of disk cylinders per cylinder group. This number must be in the range 1 to 32. *minfree* specifies the minimum percentage of free disk space allowed. Once the file system capacity reaches this threshold, only the super-user is allowed to allocate disk blocks. The default value is 10%. If a disk does not revolve at 60 revolutions per second, the *rps* parameter may be specified. If a file system will have more or less than the average number of files the *nbpi* (number of bytes per inode) can be specified to increase or decrease the number of inodes that are created. Space or time optimization preference can be specified with *opt* values of "s" for space or "t" for time. Users with special demands for their file systems are referred to the paper cited below for a discussion of the tradeoffs in using different configurations.

**SEE ALSO**

*fsck(1FFS), newfs(1FFS), tunefs(1FFS), dumpfs(1FFS)*
*fs(4FFS), dir(4FFS)* in the *Programmer's Reference Manual.*

M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems 2*, 3. pp 181-197, August 1984 (reprinted in the System Manager's Manual, SMM:14).

**ERRORS**

There should be some way to specify bad blocks.

**NAME**

mkfs.s51k – construct a file system

**SYNOPSIS**

**/etc/mkfs.s51k** *special blocks* [ *:i-nodes* ] [ *gap blocks/cyl* ]
**/etc/mkfs.s51k** *special proto* [*gap blocks/cyl*]

**DESCRIPTION**

*mkfs.s51k* constructs a file system by writing on the *special* file using the values found in the remaining arguments of the command line. The command waits 10 seconds before starting to construct the file system. During this 10-second pause the command can be aborted by entering a delete (DEL).

If the second argument is a string of digits, the size of the file system is the value of *blocks* interpreted as a decimal number. This is the number of *physical* (512 byte) disk blocks the file system will occupy. If the number of i-nodes is not given, the default is the number of *logical* (1024 byte) blocks divided by 4. *mkfs.s51k* builds a file system with a single empty directory on it. The boot program block (block zero) is left uninitialized.

If the second argument is the name of a file that can be opened, **mkfs.s51k** assumes it to be a prototype file *proto*, and will take its directions from that file. The prototype file contains tokens separated by spaces or new-lines. A sample prototype specification follows (line numbers have been added to aid in the explanation):

```
1.    /stand/ diskboot
2.    4872 110
3.    d--777 3 1
4.    usr     d--777 3 1
5.            sh      ---755 3 1 /bin/sh
6.            ken     d--755 6 1
7.                    $
8.            b0      b--644 3 1 0 0
9.            c0      c--644 3 1 0 0
10.           $
11.   $
```

Line 1 in the example is the name of a file to be copied onto block zero as the bootstrap program.

Line 2 specifies the number of *physical* (512 byte) blocks the file system is to occupy and the number of i-nodes in the file system.

Lines 3-9 tell **mkfs.s51k** about files and directories to be included in this file system.

Line 3 specifies the root directory.

lines 4-6 and 8-9 specifies other directories and files.

The $ on line 7 tells **mkfs.s51k** to end the branch of the file system it is on, and continue from the next higher directory. The $ on lines 10 and 11 end the process, since no additional specifications follow.

File specifications give the mode, the user ID, the group ID, and the initial contents of the file. Valid syntax for the contents field depends on the first character of the mode.

The mode for a file is specified by a 6-character string. The first character specifies the type of the file. The character range is *−bcd* to specify regular, block special, character special and directory files respectively. The second character of the mode is either *u* or − to specify set-user-id mode or not. The third is *g* or − for the set-group-id mode. The rest of the mode is a 3 digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod (1)*).

Two decimal number tokens come after the mode; they specify the user and group IDs of the owner of the file.

If the file is a regular file, the next token of the specification may be a path name whence the contents and size are copied. If the file is a block or character special file, two decimal numbers follow which give the major and minor device numbers. If the file is a directory, **mkfs.s51k** makes the entries . and .. and then reads a list of names and (recursively) file specifications for the entries in the directory. As noted above, the scan is terminated with the token **$**.

The final argument in both forms of the command specifies the rotational *gap* and the number of *blocks/cyl*. If the *gap* and *blocks/cyl* are not specified or are considered illegal values a default value of gap size 14 and 630 blocks/cyl is used.

**FILES**

*/etc/vtoc/**

**SEE ALSO**

*chmod(1)* in the *User's Reference Manual.*
*dir(4)* in the *Programmer's Reference Manual.*

**ERRORS**

With a prototype file, it is not possible to copy in a file larger than 64K bytes, nor is there a way to specify links. The maximum number of i-nodes configurable is 65500.

**NAME**

mknod – build special file

**SYNOPSIS**

**/etc/mknod** *name* **b** | **c** *major minor*
**/etc/mknod** *name* **p**

**DESCRIPTION**

*mknod* makes a directory entry and corresponding i-node for a special file.

The first argument is the *name* of the entry. The UNIX System convention is to keep such files in the /dev directory.

In the first case, the second argument is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g., unit, drive, or line number). They may be either decimal or octal. The assignment of major device numbers is specific to each system. The information is contained in the system source file *conf.c*. You must be the super-user to use this form of the command.

The second case is the form of the **mknod** that is used to create FIFO's (a.k.a named pipes).

**WARNING**

If **mknod** is used to create a device in a remote directory (Remote File Sharing), the major and minor device numbers are interpreted by the server.

**SEE ALSO**

*mknod(2)* in the *Programmer's Reference Manual.*

**NAME**

mkpdata.ffs – build file for mkproto

**SYNOPSIS**

**/etc/mkpdata.ffs**

**DESCRIPTION**

*mkpdata.ffs* creates a prototype filesystem file from the contents of the tree with the current directory as its root. The resulting prototype file can be used with *mkproto(1FFS)* to build a prototype filesystem.

**SEE ALSO**

*mkproto(1FFS).*

**NAME**

mkproto.ffs – construct a prototype file system

**SYNOPSIS**

**/etc/mkproto.ffs** *special proto*

**DESCRIPTION**

*mkproto.ffs* is used to bootstrap a new file system. First a new file system is created using *newfs.ffs(1FFS)*. *mkproto.ffs* is then used to copy files from the old file system into the new file system according to the directions found in the prototype file *proto*. The prototype file contains tokens separated by spaces or new lines. The first tokens comprise the specification for the root directory. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **–bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or **–** to specify set-user-id mode or not. The third is **g** or **–** for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod(1)*).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, **mkproto.ffs** makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

A sample prototype specification follows:

```
d--777 3 1
usr     d--777 3 1
        sh      ---755 3 1 /bin/sh
        ken     d--755 6 1
                $
        b0      b--644 3 1 0 0
        c0      c--644 3 1 0 0
        $
$
```

**SEE ALSO**

*fsck(1FFS), newfs(1FFS)*
*fs(4FFS), dir(4FFS)* in the *Programmer's Reference Manual.*

**BUGS**

There should be some way to specify links.

There should be some way to specify bad blocks.

*mkproto.ffs* can only be run on virgin file systems. It should be possible to copy files into existent file systems.

**NAME**

mount, umount – mount and dismount filesystems

**SYNOPSIS**

**/etc/mount** [ **–p** ]
**/etc/mount –a**[**cfv**] [ **–t** *type* ]
**/etc/mount** [ **–cfrv** ] [ **–t** *type* ] [ **–o** *options* ] *fsname dir*
**/etc/mount** [ **–cfv** ] *fsname* | *dir*

**/etc/umount** [ **–h** *host* ] [ **–frv** ]
**/etc/umount –a**[**v**]
**/etc/umount** [ **–v** ]
**/etc/umount** [ **–t** *type* ]

**DESCRIPTION**

*mount* announces to the system that a filesystem *fsname* is to be attached to the file tree at the directory *dir*. The directory *dir* must already exist. It becomes the name of the newly mounted root. The contents of *dir* are hidden until the filesystem is unmounted. If *fsname* is of the form host:path the filesystem type is assumed to be *nfs(4)*.

*umount* announces to the system that the filesystem *fsname* previously mounted on directory *dir* should be removed. Either the filesystem name or the mounted-on directory may be used.

*mount* and *umount* maintain a table of mounted filesystems in */etc/mtab,* described in *mtab(4)*. If invoked without an argument, **mount** displays the table. If invoked with only one of *fsname* or *dir* **mount** searches the file */etc/fstab* (see *fstab(4)*) for an entry whose *dir* or *fsname* field matches the given argument. For example, if this line is in */etc/fstab*:

"/dev/xy0g /usr ffs rw 1 1"

then the commands *mount /usr* and *mount /dev/xy0g* are shorthand for *mount /dev/xy0g /usr* .

**MOUNT OPTIONS**

**–p** Print the list of mounted filesystems in a format suitable for use in */etc/fstab*.

**–a** Attempt to mount all the filesystems described in */etc/fstab*. (In this case, *fsname* and *dir* are taken from */etc/fstab*.) If a type is specified all of the filesystems in */etc/fstab* with that type are mounted. Filesystems are not necessarily mounted in the order listed in */etc/fstab* .

**–c** Invoke *fsstat(1M)* on each filesystem being mounted, and if it indicates that the filesystem is dirty, call *fsck(1M)* to clean the filesystem. *fsck* is passed the *-D* and *-y* options.

**–f** Fake a new */etc/mtab* entry, but do not actually mount any filesystems.

**–v** Verbose – **mount** displays a message indicating the filesystem being mounted.

**–t** The next argument is the filesystem type. The accepted types are **ffs**, **s51k**, and **nfs**; see *fstab*(4) for a description of these filesystem types.

**–r** Mount the specified filesystem read-only. This is a shorthand for:

**mount –o ro** *fsname dir*

Physically write-protected and magnetic tape filesystems must be mounted read-only, or errors occur when access times are updated, whether or not any explicit write is attempted.

**–o** Specify *options,* a list of comma-separated words from the list below. Some options are valid for all filesystem types, while others apply to a

specific type only.

*options* valid on *all* file systems (the defaults are **rw,suid,fsck**):

**rw** read/write.

**ro** read-only.

**suid** set-uid execution allowed.

**nosuid** set-uid execution not allowed.

**suid** and **nosuid** are currently not supported.

**raw=path**
the filesystem's raw device interface pathname.

**fsck** *fsck(1M)* invoked with no filesystem arguments should check this filesystem.

**nofsck** *fsck(1M)* should not check this filesystem by default.

**hide** ignore this entry during a **mount -a** command to allow you to define *fstab* entries for commonly used filesystems you don't want to automatically mount.

*options* specific to **nfs** (NFS) file systems (the defaults are:

**fg,retry=0,timeo=7,retrans=4,port=NFS_PORT,hard**

with defaults for *rsize* and *wsize* set by the kernel):

**bg** if the first mount attempt fails, retry in the background.

**fg** retry in foreground.

**retry=***n* set number of mount failure retries to *n*.

**rsize=***n* set read buffer size to *n* bytes.

**wsize=***n* set write buffer size to *n bytes.*

**timeo=***n* set NFS timeout to *n* tenths of a second.

**retrans=***n* set number of NFS retransmissions to *n*.

**port=***n* set server IP port number to *n*.

**soft** return error if server doesn't respond.

**hard** retry request until server responds.

The **bg** option causes **mount** to run in the background if the server's *mountd(1M)* does not respond. If **bg** is specified and retry is not specified, **retry** defaults to 10000. *mount* attempts each request **retry=***n* times before giving up. Once the filesystem is mounted, each NFS request made in the kernel waits **timeo= n** tenths of a second for a response. If no response arrives, the time-out is multiplied by **2** and the request is retransmitted. When **retrans=***n* retransmissions have been sent with no reply a **soft** mounted filesystem returns an error on the request and a **hard** mounted filesystem retries the request. Filesystems that are mounted **rw** (read-write) should use the **hard** option. The number of bytes in a read or write request can be set with the **rsize** and **wsize** options.

**UMOUNT OPTIONS**

**-h** *host*
Unmount all filesystems listed in */etc/mtab* that are remote-mounted from *host*.

**-a** Attempt to unmount all the filesystems currently mounted (listed in */etc/mtab*). In this case, *fsname* is taken from */etc/mtab*.

**-t** Unmounts all filesystems of a given filesystem type. The accepted types are **ffs**, **s51k**,

and **nfs**.

**−v** Verbose – *umount* displays a message indicating the filesystem being unmounted.

**EXAMPLES**

| | |
|---|---|
| mount /dev/xy0g /usr | **mount a local disk** |
| **mount −at ffs** | mount all ffs filesystems |
| mount −t nfs serv:/usr/src /usr/src | **mount remote filesystem** |
| **mount serv:/usr/src /usr/src** | same as above |
| mount −o hard serv:/usr/src /usr/src | **same as above but hard mount** |
| **mount −p > /etc/fstab** | save current mount state |

**FILES**

| | |
|---|---|
| */etc/mtab* | mount table |
| */etc/fstab* | filesystem table |

**SEE ALSO**

*nfsmount(2), fstab(4), mountd(1M), nfsd(1M)*

**ERRORS**

Mounting filesystems full of garbage crashes the system.

If the directory on which a filesystem is to be mounted is a symbolic link, the filesystem is mounted on *the directory to which the symbolic link refers,* rather than being mounted on top of the symbolic link itself.

**ORIGIN**

Sun Microsystems

**NAME**

mountall, umountall – mount, unmount multiple file systems

**SYNOPSIS**

**/etc/mountall** [**–**] [*file-system-table*] **...**
**/etc/umountall** [ **–k** ]

**DESCRIPTION**

These commands may be executed only by the super-user.

*mountall* is used to mount file systems according to a *file-system-table*. (*/etc/fstab* is the default file system table.) The special file name "–" reads from the standard input.

Before each file system is mounted, it is checked using *fsstat(1M)* to see if it appears mountable. If the file system does not appear mountable, it is checked, using *fsck(1M)*, before the mount is attempted.

*umountall* causes all mounted file systems except *root* to be unmounted. The **–k** option sends a SIGKILL signal, via *fuser(1M)*, to processes that have files open.

**FILES**

File-system-table format:

| | |
|---|---|
| column 1 | block special file name of file system |
| column 2 | mount-point directory |
| column 3 | "–r" if to be mounted read-only; "–d" if remote |
| column 4 | (optional) file system type string |
| column 5+ | ignored |

White-space separates columns. Lines beginning with "#" are comments. Empty lines are ignored.

A typical file-system-table might read:

```
/dev/dsk/c1d0s2    /usr -r S51K
```

**SEE ALSO**

*fsck(1M), fsstat(1M), fuser(1M), mount(1M).*
*sysadm(1)* in the *User's Reference Manual.*
*signal(2), fstab(4)* in the *Programmer's Reference Manual.*

**DIAGNOSTICS**

No messages are printed if the file systems are mountable and clean.

Error and warning messages come from *fsck(1M)*, *fsstat1(1M)*, and *mount(1M)*.

**NAME**

mountd – NFS mount request server

**SYNOPSIS**

**/usr/etc/rpc.mountd**

**DESCRIPTION**

*mountd* is an *rpc(4)* server that answers file system mount requests. It reads the file */etc/exports*, described in *exports(4)*, to determine which file systems are available to which machines and users. It also provides information as to which clients have file systems mounted. This information can be printed using the *showmount(1M)* command.

The **mountd** daemon is normally invoked by *inetd(1M)*.

**SEE ALSO**

*exports(4), services(4), inetd(1M), showmount(1M)*

**ORIGIN**

Sun Microsystems

**NAME**

mvdir – move a directory

**SYNOPSIS**

**/etc/mvdir** *dirname name*

**DESCRIPTION**

*mvdir* moves directories within a file system. *dirname* must be a directory. If *name* does not exist, it will be created as a directory. If *name* does exist, *dirname* will be created as *name/dirname*. *dirname* and *name* may not be on the same path; that is, one may not be subordinate to the other. For example:

mvdir x/y x/z

*is legal, but*

*mvdir x/y x/y/z*

is not.

**SEE ALSO**

*mkdir(1), mv(1)* in the *User's Reference Manual.*

**WARNINGS**

Only the super-user can use **mvdir.**

**NAME**

ncheck.ffs – generate names from i-numbers

**SYNOPSIS**

**/etc/ncheck.ffs** [ **–i** *numbers* ] [ **–a** ] [ **–s** ] *filesystems* ...

**DESCRIPTION**

**N.B.:** For most normal file system maintenance, the function of *ncheck.ffs* is subsumed by *fsck(1M)*.

*ncheck* with no options generates a pathname vs. i-number list of all files on every specified filesystem. Names of directory files are followed by '/.'. The **–i** option reduces the report to only those files whose i-numbers follow. The **–a** option allows printing of the names '.' and '..', which are ordinarily suppressed. The **–s** option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

The report is in no useful order, and probably should be sorted.

**SEE ALSO**

*sort(1), fsck(1M)*

**DIAGNOSTICS**

When the filesystem structure is improper, '??' denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

**NAME**

ncheck – generate path names from i-numbers

**SYNOPSIS**

**/etc/ncheck** [ **–i** *i-numbers* ] [ **–a** ] [ **–s** ] [ *file-system* ]

**DESCRIPTION**

*ncheck* with no arguments generates a path-name vs. i-number list of all files on a set of default file systems (see */etc/checklist*). Names of directory files are followed by **/..**

The options are as follows:

**–i** limits the report to only those files whose i-numbers follow.

**–a** allows printing of the names **.** and **..**, which are ordinarily suppressed.

**–s** limits the report to special files and files with set-user-ID mode. This option may be used to detect violations of security policy.

*file system* must be specified by the file system's special file.

The report should be sorted so that it is more useful.

**SEE ALSO**

*fsck(1M).*
*sort(1)* in the *User's Reference Manual.*

**DIAGNOSTICS**

If the file system structure is not consistent, **??** denotes the "parent" of a parentless file and a path-name beginning with **...** denotes a loop.

**NAME**

**netstat** - show network status

**SYNOPSIS**

**netstat** [ **−Aan** ] [ **−f** *address_family* ] [ *system* ] [ *core* ]
**netstat** [ **−himnrs** ] [ **−f** *address_family* ] [ *system* ] [ *core* ]
**netstat** [ **−n** ] [ **−I** *interface* ] *interval* [ *system* ] [ *core* ]

**DESCRIPTION**

The *netstat* command symbolically displays the contents of various network-related data structures. There are a number of output formats, depending on the options for the information presented. The first form of the command displays a list of active sockets for each protocol. The second form presents the contents of one of the other network data structures according to the option selected. Using the third form, with an *interval* specified, *netstat* will continuously display the information regarding packet traffic on the configured network interfaces.

The options have the following meaning:

**−A** With the default display, show the address of any protocol control blocks associated with sockets; used for debugging.

**−a** With the default display, 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).

**−I** *interface*
Show information only about this interface; used with an *interval* as described below.

**−m** Show statistics recorded by the memory management routines (the network manages a private pool of memory buffers).

**−n** Show network addresses as numbers (normally **netstat** interprets addresses and attempts to display them symbolically). This option may be used with any of the display formats.

**−s** Show per-protocol statistics.

**−r** Show the routing tables. When **−s** is also present, show routing statistics instead.

**−f** *address_family*
Limit statistics or address control block reports to those of the specified *address family*. The following address families are recognized: *inet*, for **AF_INET**, *ns*, for **AF_NS**, and *unix*, for **AF_UNIX**.

The arguments, *system* and *core* allow substitutes for the defaults "/vmunix" and "/dev/kmem". **AF_UNIX** is unsupported.

The default display, for active sockets, shows the local and remote addresses, send and receive queue sizes (in bytes), protocol, and 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 */etc/hosts* and */etc/networks*, respectively. If a symbolic name for an address is unknown, or if the **−n** option is specified, the address is printed numerically, according to the address family. For more information regarding the Internet "dot format," refer to *inet(3N)*. 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 addresses of the interface and the maximum transmission unit ("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"), whether the route is to a gateway ("G"), and whether the route was created dynamically by a redirect ("D"). Direct routes are created for each interface attached to the local host; the gateway field for such entries shows the address of the outgoing interface. 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 while sending to the same destination. 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 for the primary interface (the first interface found during autoconfiguration) and a column summarizing information for all interfaces. The primary interface may be replaced with another interface with the *−I* option. 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**

*iostat(1), vmstat(1), hosts(4), networks(4), protocols(4), services(4)*

**ERRORS**

The notion of errors is ill-defined. Collisions mean something else for the IMP.

**ORIGIN**

4.3 BSD

**NAME**

newfs.ffs - construct a new file system

**SYNOPSIS**

**/etc/newfs.ffs** [ **−N** ] [ **−v** ] [ **−n** ] [ *mkfs.ffs-options* ] *special disk-type*

**DESCRIPTION**

*newfs.ffs* is a "friendly" front-end to the *mkfs.ffs(1M)* program. **newfs.ffs** will look up the type of disk a file system is being created on in the disk description file */etc/disktab*, calculate the appropriate parameters to use in calling *mkfs.ffs*, then build the file system by forking *mkfs.ffs* and, if the file system is a root partition, install the necessary bootstrap programs in the initial 8 sectors of the device. The **−n** option prevents the bootstrap programs from being installed. (In UMIPS-V, there are no bootstrap programs, so the **−n** option is allowed but has no effect.) The **−N** option causes the file system parameters to be printed out without actually creating the file system.

If the **−v** option is supplied, **newfs.ffs** will print out its actions, including the parameters passed to *mkfs.ffs*.

Options which may be used to override default parameters passed to *mkfs.ffs* are:

**-s** *size* — The size of the file system in sectors. This number will be rounded down to an integral number of filesystem blocks in order to properly create the filesystem.

**-b** *block-size* — The block size of the file system in bytes.

**-f** *frag-size* — The fragment size of the file system in bytes.

**-t** *#tracks/cylinder*

**-c** *#cylinders/group* — The number of cylinders per cylinder group in a file system. The default value used is 16.

**-m** *free space %* — The percentage of space reserved from normal users; the minimum free space threshhold. The default value used is 10%.

**-o** *optimization preference ("space" or "time")* — The file system can either be instructed to try to minimize the time spent allocating blocks, or to try to minimize the space fragmentation on the disk. If the value of minfree (see above) is less than 10%, the default is to optimize for space; if the value of minfree greater than or equal to 10%, the default is to optimize for time.

**-r** *revolutions/minute* — The speed of the disk in revolutions per minute (normally 3600).

**-S** *sector-size* — The size of a sector in bytes (almost never anything but 512).

**-i** *number of bytes per inode* — This specifies the density of inodes in the file system. The default is to create an inode for each 2048 bytes of data space. If fewer inodes are desired, a larger number should be used; to create more inodes a smaller number should be given.

**FILES**

| | |
|---|---|
| */etc/disktab* | for disk geometry and file system partition information |
| */etc/mkfs.ffs* | to actually build the file system |

**SEE ALSO**

*disktab(5), fs(4FFS), fsck(1FFS), mkfs(1FFS), tunefs(1FFS), dumpfs(1FFS)*

M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems 2*, 3. pp 181-197, August 1984 (reprinted in the System Manager's Manual, SMM:14).

**ERRORS**

Should figure out the type of the disk without the user's help.

**NAME**

newgrp – log in to a new group

**SYNOPSIS**

**newgrp** [ **–** ] [ *group* ]

**DESCRIPTION**

*newgrp* changes a user's group identification. The user remains logged in and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new real and effective group IDs. The user is always given a new shell, replacing the current shell, by *newgrp,* regardless of whether it terminated successfully or due to an error condition (i.e., unknown group).

Exported variables retain their values after invoking *newgrp;* however, all unexported variables are either reset to their default value or set to null. System variables (such as PS1, PS2, PATH, MAIL, and HOME), unless exported by the system or explicitly exported by the user, are reset to default values. For example, a user has a primary prompt string (PS1) other than $ (default) and has not exported PS1. After an invocation of *newgrp,* successful or not, their PS1 will now be set to the default prompt string $. Note that the shell command *export* (see *sh(1)*) is the method to export variables so that they retain their assigned value when invoking new shells.

With no arguments, *newgrp* changes the group identification back to the group specified in the user's password file entry. This is a way to exit the effect of an earlier *newgrp* command.

If the first argument to *newgrp* is a **–**, the environment is changed to what would be expected if the user actually logged in again as a member of the new group.

A password is demanded if the group has a password and the user does not, or if the group has a password and the user is not listed in */etc/group* as being a member of that group.

**FILES**

| | |
|---|---|
| */etc/group* | system's group file |
| */etc/passwd* | system's password file |

**SEE ALSO**

*login(1), sh(1)* in the *User's Reference Manual.*
*group(4), passwd(4), environ(5)* in the *Programmer's Reference Manual.*

**ERRORS**

There is no convenient way to enter a password into */etc/group*. Use of group passwords is not encouraged, because, by their very nature, they encourage poor security practices. Group passwords may disappear in the future.

**NAME**

nfsd, biod – NFS daemons

**SYNOPSIS**

**/etc/nfsd** [*nservers*]

**/etc/biod** [*nservers*]

**DESCRIPTION**

*nfsd* starts the *nfs(4)* server daemons that handle client filesystem requests. *nservers* is the number of file system request daemons to start. This number should be based on the load expected on this server. Four seems to be a good number.

**biod** starts *nservers* asynchronous block I/O daemons. This command is used on a NFS client to buffer cache handle read-ahead and write-behind. The magic number for *nservers* in here is also four.

**SEE ALSO**

*mountd(1M), exports(4)*

**ORIGIN**

Sun Microsystems

**NAME**

nfsstat – Network File System statistics

**SYNOPSIS**

**nfsstat** [ **–csnrdz** ] [ *unix* ] [ *core* ]

**DESCRIPTION**

*nfsstat* displays statistical information about the Network File System (NFS) and Remote Procedure Call (RPC) interfaces to the kernel. It can also be used to reinitialize this information. If no options are given the default is

nfsstat –csnr

That is, print everything and reinitialize nothing. The optional arguments *unix* and *core* may be used to indicate another system namelist and kernel memory image, respectively.

**OPTIONS**

| | |
|---|---|
| **–c** | Display client information. Only the client side NFS and RPC information will be printed. Can be combined with the **–n** and **–r** options to print client NFS or client RPC information only. |
| **–s** | Display server information. Works like the **–c** option above. |
| **–n** | Display NFS information. NFS information for both the client and server side will be printed. Can be combined with the **–c** and **–s** options to print client or server NFS information only. |
| **–r** | Display RPC information. Works like the **–n** option above. |
| **–z** | Zero (reinitialize) statistics. Can be combined with any of the above options to zero particular sets of statistics after printing them. |
| **–d** | The user must have write permission on */dev/kmem* for this option to work. |

**FILES**

| | |
|---|---|
| */unix* | system namelist |
| */dev/kmem* | kernel memory |

**ORIGIN**

Sun Microsystems

**NAME**

p – put the contents of a memory location or machine register

**SYNOPSIS**

**p** [ *width* ] *address value*

**DESCRIPTION**

The *put* command sets the contents of a single memory location (*addressR) to value*.

When you use the *put* command with the debug monitor, *address* may be a general, special, or system coprocessor register name. The general purpose registers may be specified as either *r0* through *r31*, or by the compiler-usage names. See *dbgmon*(1spp) for more details on the general and special registers.

The default for width is word. An alternate width can be selected by entering one of the following characters on the command line as an argument:

**-b** Byte (8 bits)

**-h** Halfword (16 bits)

**-w** Word (32 bits)

**SEE ALSO**

g(1prom), dump(1prom), fill(1prom), dbgmon(1spp)

**NAME**

periodic: hourly, daily, weekly, monthly – periodic administration interface

**SYNOPSIS**

**/usr/adm/periodic/driver** *period*

**DESCRIPTION**

This command is run by *cron(1m)* from the *periodic* crontab file in order to provide a simple interface for executing hourly, daily, weekly, and monthly system tasks. The execution by *cron* sends the output of the scripts, except for hourly runs, to the user *root* by mail, providing a record of the tasks.

The *period* argument must be either **hourly**, **daily**, **weekly**, or **monthly**.

Executing the driver causes execution of some or all of the task files in the directory /usr/adm/periodic/*period*. Only files whose names begin with a digit and end with either **.system**, **.local**, or *.hostname* (where *hostname* is the name of the host executing the driver) are candidates for execution. If there is a file whose name is the same as a candidate with the additional suffix **.stop**, execution is held, and the contents of the **.stop** file is printed (to document the reason for its existence).

The files are executed in alphabetically sorted order. For example, the file /usr/adm/periodic/daily/01.clean.system would be executed before /usr/adm/periodic/daily/10.mail.system.

Execution is done with a *nice(1)* value of 15. This will allow things such as UUCP jobs and jobs executed by real users working late at night to run more reasonably. If a periodic script needs to be run at a higher priority, this should be done inside of the script.

The intent of this interface is to provide Mips with a means of adding system periodic functions without having to add them to the crontab file, which may be overwritten by the system administrator without them even knowing it.

In a network of machines, this system of suffixes provides some flexibility. Specifically, all of the tasks to be executed can be maintained on one system, using the **.local** files as network-wide commands, using the host-specific files for host-specific operations, and distributing all files throughout the network.

Administrators writing their own scripts or programs for this interface should note that no environment, PATH, or current working directory should be assumed.

It is recommended that **.system.stop** files be used temporarily or for report-only tasks, and not for permanent disabling of Mips-provided tasks, as Mips may make important changes. If a task does need to be disabled, it is a good idea to inform the system supplier of this need.

**SEE ALSO**

*nice(1), cron(1m).*

**NAME**

ping – send ICMP ECHO_REQUEST packets to network hosts

**SYNOPSIS**

**/etc/ping** [ **–r** ] [ **–v** ] *host* [ *packetsize* ] [ *count* ]

**DESCRIPTION**

The DARPA Internet is a large and complex aggregation of network hardware, connected together by gateways. Tracking a single-point hardware or software failure can often be difficult. *ping* utilizes the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams ("pings") have an IP and ICMP header, followed by a **struct timeval**, and then an arbitrary number of "pad" bytes used to fill out the packet. Default datagram length is 64 bytes, but this may be changed using the command-line option. Other options are:

**–r** Bypass the normal routing tables and send directly to a host on an attached network. If the host is not on a directly-attached network, an error is returned. This option can be used to ping a local host through an interface that has no route through it (e.g., after the interface was dropped by *routed(1M)* ).

**–v** Verbose output. ICMP packets other than ECHO_RESPONSE that are received are listed.

When using *ping* for fault isolation, it should first be run on the local host, to verify that the local network interface is up and running. Then, hosts and gateways further and further away should be "pinged". *ping* sends one datagram per second, and prints one line of output for every ECHO_RESPONSE returned. No output is produced if there is no response. If an optional *count* is given, only that number of requests is sent. Round-trip times and packet loss statistics are computed. When all responses have been received or the program times out (with a *count* specified), or if the program is terminated with a SIGINT, a brief summary is displayed.

This program is intended for use in network testing, measurement and management. It should be used primarily for manual fault isolation. Because of the load it could impose on the network, it is unwise to use *ping* during normal operations or from automated scripts.

**AUTHOR**

Mike Muuss

**SEE ALSO**

*netstat(1), ifconfig(1M)*

**NAME**

portmap – DARPA port to RPC program number mapper

**SYNOPSIS**

**/usr/etc/rpc.portmap**

**DESCRIPTION**

*portmap* is a server that converts RPC program numbers into DARPA protocol port numbers. It must be running in order to make RPC calls.

When an RPC server is started, it will tell *portmap* what port number it is listening to, and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact *portmap* on the server machine to determine the port number where RPC packets should be sent.

Normally, standard RPC servers are started by *inetd(1M)*, so *portmap* must be started before *inetd* is invoked.

**SEE ALSO**

*rpcinfo(1M)*, *inetd(1M)*

**ERRORS**

If *portmap* crashes, all servers must be restarted.

**ORIGIN**

Sun Microsystems

**NAME**

powerdown – stop all processes and turn off the power

**SYNOPSIS**

*powerdown* [ **–y** | **–Y** ]

**DESCRIPTION**

This command brings the system to a state where nothing is running and then turns off the power.

By default, the user is asked questions that control how much warning the other users are given. The options:

**–y** prevents the questions from being asked and just gives the warning messages. There is a 60 second pause between the warning messages. Note that pressing the standby button on the side of the cabinet will accomplish the same thing.

**–Y** is the same as **–y** except it has no pause between messages. It is the fastest way to bring the system down.

The identical function is also available under the *sysadm* command:

sysadm powerdown

Password control can be instituted on this command. See *sysadm(1), admpasswd* subcommand.

**EXAMPLES**

some-long-running-command; powerdown –y

The first command is run to completion and then the machine turns off. This is useful for, say, formatting a document to the printer at the end of a day.

**FILES**

*/etc/shutdown* - invoked by powerdown

**SEE ALSO**

*shutdown(1M).*
*sysadm(1)* in the *User's Reference Manual.*

**NAME**

pr_tod – prints contents of time-of-day register

**SYNOPSIS**

**pr_tod**

**DESCRIPTION**

This command prints the contents of the time-of-day register for the time-of-day chip. You can determine whether the time-of-day chip is functioning by running this command twice. If you run the command and then run it again 5 or 6 seconds later, then the value displayed the second time should be about 5 or 6 more that the first time. If the displayed value does not change or if the value decreases, then run the *init_tod* command to correct this situation.

**SEE ALSO**

init_tod(1prom)

NAME

printenv – display prom environment variables

SYNOPSIS

**printenv** [ varlist ]

DESCRIPTION

*printenv* displays the value of the PROM Monitor variables in *varlist* . The *varlist* argument can be one or more variable separated by a space. If no environment variables are specified, *printenv* shows all currently defined variables.

SEE ALSO

setenv(1prom), unsetenv(1prom), intro(1spp)

**NAME**

profiler: prfld, prfstat, prfdc, prfsnap, prfpr – UNIX system profiler

**SYNOPSIS**

**/etc/prfld** [ *system_namelist* ]
**/etc/prfstat on**
**/etc/prfstat off**
**/etc/prfdc** *file* [ *period* [ *off_hour* ] ]
**/etc/prfsnap** *file*
**/etc/prfpr** *file* [ *cutoff* [ *system_namelist* ] ]

**DESCRIPTION**

*prfld* , *prfstat* , *prfdc* , *prfsnap* , and *prfpr* form a system of programs to facilitate an activity study of the UNIX operating system.

*prfld* is used to initialize the recording mechanism in the system. It generates a table containing the starting address of each system subroutine as extracted from *system_namelist*.

*prfstat* is used to enable or disable the sampling mechanism. Profiler overhead is less than 1% as calculated for 500 text addresses. *prfstat* will also reveal the number of text addresses being measured.

*prfdc* and *prfsnap* perform the data collection function of the profiler by copying the current value of all the text address counters to a file where the data can be analyzed. *prfdc* will store the counters into *file* every *period* minutes and will turn off at *off_hour* (valid values for *off_hour* are *0 – 24* ). *prfsnap* collects data at the time of invocation only, appending the counter values to *file*.

*prfpr* formats the data collected by *prfdc* or *prfsnap* . Each text address is converted to the nearest text symbol (as found in *system_namelist*) and is printed if the percent activity for that range is greater than *cutoff*.

**FILES**

| | |
|---|---|
| */dev/prf* | interface to profile data and text addresses |
| */unix* | default for system namelist file |

**NAME**

prom – general features of the prom monitor

**DESCRIPTION**

This man page describes the PROM Monitor. The PROM Monitor provides the tools for examining and changing PROM memory, downloading programs over serial lines (RS-232C), and booting programs from disk, tape or Ethernet. The PROM Monitor also provides tools for altering configuration power-up options in non-volatile RAM

The PROM Monitor resides in PROM on the CPU board and is entered when the system is reset or the system is powered up. The PROM Monitor initializes the processor, the memory boards, and the CPU board. The processor is initialized by initializing the system coprocessor Status and Cause registers, and flushing the translation buffer.

The memory cards are initialized by probing to determine how many cards exist, determining the best memory interleave configuration, configuring the boards for refresh slot assignment and assigning base addresses.

The CPU board is initialized by sizing and flushing the instruction and data caches, by inspecting the contents of non-volatile memory and reinitializing it if necessary, and by initializing environment variables from non-volatile memory.

**MEMORY USAGE**

The PROM Monitor uses system memory between physical address 0x500 and 0x10000. The include file "prom/entrypt.h" describes conventions for memory use by standalone programs.

**FILE NAME SYNTAX**

When the PROM Monitor requires a filename, the components should be listed for the different devices in the order shown below.

| | |
|---|---|
| **disk devices:** | device(controller, unit, partition)path |
| **SCSI tape:** | scsi_tape_device(controller, unit, file)path |
| **ethernet devices:** | boot_protocol(controller, ethernet)path |

controller — Since a system can have more that one device controller of a particular type installed, the controller field indicates the specific controller. The system defines control register addresses for multiple controllers for the system. If you do not specify a controller number, the default value of 0 is used.

device — The device field associates the file name with a particular type of hardware.

ethernet — ethernet is the name of a UDDP/IP device. (cmc or egl are the only devices correctly supported.)

file — File is the file number on the SCSI tape.

partition — Disk devices are frequently broken down into logical sub-units, called partitions. The partition field selects a disk partition within a unit. The partition's base cylinder and size is determined by accessing the disk volume header stored on the disk itself. If you do not specify the partition field, the default value of 0 is used. For tape drives the partition number specifies the file number on the tape. Numbering starts at zero; zero is the first file on the tape; 1 specifies the second file on the tape, and so on.

path — The path indicates a particular file on the media specified by the device, controller, unit, and partition fields. The file referred to by path is located by consulting a directory located on the device itself. The PROM Monitor does

not include code that locates files. However, the sash does have code to locate files from disk volume headers, tape directories, and disk directories. If you want to use this option, then boot sash first using the boot command. If you do not specify a path, then the file name is assumed to refer to the raw device.

unit — Because you can attach multiple storage devices to a single device controller, the unit number indicates the specific device. If you do not specify a unit num ber, then the default value of 0 is used. If you are using the tqij device, then the default value is 6.

**ENVIRONMENT VARIABLES**

The PROM Monitor maintains "environment variables" that are passed to booted programs. These variables function like UNIX system shell environment variables. Some of the environment variables affect the operation of the PROM Monitor and are maintained in non-volatile memory. This means that when you reset the machine or power it down, the Monitor still maintains these variables.

lbaud — Specifies the baud rate for tty(0), which is uart A on the CPU board and typically the local console. You can set the baud rate to: 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, or 19200. If you specify an illegal baud rate, 9600 baud is used.

rbaud — Specifies the baud rate for tty(1), which is uart B on the CPU board and typically the remote console. You can set the baud rate to: 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, or 19200. If you specify an illegal baud rate, 9600 baud is used.

netaddr — Specifies the internet address for the node. This is used by the *bootfile* service software in the standalone I/O (saio) library.

console — This variable selects which console devices are to be considered consoles on power-up and after resets. You can enable and disable consoles by command after reset, see *enable*(1). When set to 'l' (the letter L), only tty(0) is initially enabled as a console, if console is 'r', then both tty(0) and tty(1) are enabled as consoles.

M/2030 console variables

The console variables and the devices that they enable are given below:

| | |
|---|---|
| '0' | enables tty(0) |
| '1' | enables tty(1) |
| 'g' | enables graf(0) |
| 't' | enables tty(0) and tty(1) |
| 'a' | enables everything possible |
| 'l' | if a graphics board is in the system, then 'l' enables graf(0); otherwise it will enable tty(0), the logical console. |
| 'r' | if a graphics board is in the system, 'r' enables graf(0) and tty(0). |

bootfile — Specifies the default program that boots when you don't specify the *-f* option to the *boot* command.

bootmode — Controls the PROM Monitor's action in response to system resets. The bootmodes are given below(?).

| Bootmode | Description |
|---|---|
| m | the PROM monitor enters command mode after reset. |
| c | the PROM monitor does a cold boot. A cold boot loads the file specified by the environment variable "bootfile" and passes it the argument "-a". Typically, the bootfile is the standalone shell (sash). The standalone shell interprets the "-a" flag as a request to load the operating system as specified in the volume header of the device from which sash itself loaded. |
| w | attempts a warm boot on reset. A warm boot transfers control to an memory image loaded before you reset the system. The PROM Monitor determines if such an image is present by looking for a properly formatted "restart block", see *restart*(5spp). If a restart block is not found, a restart block is incorrectly formatted, or a warm boot has already been attempted with the restart block, then a cold boot is performed. |
| d | the PROM Monitor operates like command mode ('m') except that the Monitor attempts to preserve the contents of memory across resets. |

| | |
|---|---|
| cpuid | Reserved for future use. Currently this variable must be set to zero. |
| resetepc | This variable indicates the program counter the machine was executing when the machine was reset. |
| resetra | This variable indicates the contents of the Return Address register when the machine was reset. |
| memparity | Setting this variable to one (1) enables parity, and setting this variable to zero (0) disables parity. This variable should be used in conjunction with the kernel argument "mem_parity" to enable and disable parity when running UNIX. (Not supported on M/2030). |
| version | This variable indicates the version of the installed PROMs, and it is used by the kernel to determine which PROMs are installed in the machine. This environment variable cannot be changed. |

## INPUT EDITING

The following basic editing commands are available for the PROM Monitor.

| | |
|---|---|
| control-H or DEL | Erases the previous character. |
| Control-U | Erases the entire line. |
| Control-C | Aborts the program that is currently running and returns control to the PROM Monitor. |
| Control-Z | Causes the current program to execute a breakpoint instruction. This command is used in conjunction with the standalone program dbgmon. |
| Control-D | Causes the standalone program to exit normally. |

## USING BREAKS TO CHANGE BAUD RATE

You can also cycle the baud rate for tty(0) and tty(1) among the baud rates, 110, 300, 1200, 2400, 4800, 9600, and 19200 baud by entering a BREAK. Baud rates changes made by BREAK's are temporary until the next reset or until a new program is loaded. To change the baud rate permanently, change either the lbaud or rbaud environment variable.

## EXTENDING THE PROM MONITOR

If you give the PROM Monitor a command that is not built-in, then the Monitor uses the first word of the command as the name of a file and tries to boot that file passing any other

arguments on the command line onto the booted program. If the environment variable $path is undefined, then the first word of the command must be a complete file name specification consisting of a device name, controller, unit, partition fields as necessary, and a file path. If the environment variable $path is defined, the PROM Monitor tries to boot the program file formed by prepending the contents of $path to the command. If $path is a list of prefixes separated by spaces, then the PROM Monitor tries each prefix from $path in turn until the file boots successfully or all prefixes have been tried.

**SEE ALSO**

intro(1spp), sash(1Mspp), dbgmon(1spp)

**NAME**

prtvtoc – print the volume header of a disk

**SYNOPSIS**

**/etc/prtvtoc** [ **–f** ] [ **–h** ] [ **–s** ] [ **–t** *fstab* ] *device...*

**DESCRIPTION**

*prtvtoc* allows the contents of the VTOC1 (volume table of contents), also known as the volume header, to be viewed by the super-user for reference or verification. The command can be used only by the super-user.

The *device* name must be the file name of a raw disk device.

**OPTIONS**

| | |
|---|---|
| **–f** | Print *sh(1)* variable assignments for information regarding the free space area. |
| **–h** | Omit all headers. |
| **–s** | Omit all but the column headers. |
| **–t** *fstab* | Use the named file instead of */etc/fstab*. |

**SEE ALSO**

dvhtool(1M).

**ERRORS**

The output format is really specific to AT&T machines, and not for Mips machines.

**NAME**

pwck, grpck – password/group file checkers

**SYNOPSIS**

**/etc/pwck** [ *file* ]
**/etc/grpck** [ *file* ]

**DESCRIPTION**

*pwck* scans the password file and notes any inconsistencies. The checks include validation of the number of fields, login name, user ID, group ID, and whether the login directory and the program-to-use-as-Shell exist. The default password file is */etc/passwd*.

*grpck* verifies all entries in the group file. This verification includes a check of the number of fields, group name, group ID, and whether all login names appear in the password file. The default group file is */etc/group*.

**FILES**

*/etc/group*
*/etc/passwd*

**SEE ALSO**

*group(4), passwd(4)* in the *Programmer's Reference Manual.*

**DIAGNOSTICS**

Group entries in */etc/group* with no login names are flagged.

**NAME**

rc0 – run commands performed to stop the operating system

**SYNOPSIS**

**/etc/rc0**

**DESCRIPTION**

This file is executed at each system state change that needs to have the system in an inactive state. It is responsible for those actions that bring the system to a quiescent state, traditionally called "shutdown".

There are three system states that require this procedure. They are state 0 (the system halt state), state 5 (the firmware state), and state 6 (the reboot state). Whenever a change to one of these states occurs, the */etc/rc0* procedure is run. The entry in */etc/inittab* might read:

```
s0:056:wait:/etc/rc0 >/dev/console 2>&1 </dev/console
```

Some of the actions performed by **/etc/rc0** are carried out by files in the directory **/etc/shutdown.d**. and files beginning with **K** in **/etc/rc0.d**. These files are executed in ascii order (see FILES below for more information), terminating some system service. The combination of commands in **/etc/rc0** and files in */etc/shutdown.d* and **/etc/rc0.d** determines how the system is shut down.

The recommended sequence for **/etc/rc0** is:

Stop System Services and Daemons.

Various system services (such as 3BNET Local Area Network or LP Spooler) are gracefully terminated.

When new services are added that should be terminated when the system is shut down, the appropriate files are installed in **/etc/shutdown.d** and **/etc/rc0.d**.

Terminate Processes

SIGTERM signals are sent to all running processes by *killall(1M)*. Processes stop themselves cleanly if sent SIGTERM.

Kill Processes

SIGKILL signals are sent to all remaining processes; no process can resist SIGKILL.

At this point the only processes left are those associated with **/etc/rc0** and processes 0 and 1, which are special to the operating system.

Unmount All File Systems

Only the root file system (/) remains mounted.

Depending on which system state the systems end up in (0, 5, or 6), the entries in */etc/inittab* will direct what happens next. If the */etc/inittab* has not defined any other actions to be performed as in the case of system state 0, then the operating system will have nothing to do. It should not be possible to get the system's attention. The only thing that can be done is to turn off the power or possibly get the attention of a firmware monitor. The command can be used only by the super-user.

**FILES**

The execution by */bin/sh* of any files in */etc/shutdown.d* occurs in ascii sort-sequence order. See *rc2(1M)* for more information.

**SEE ALSO**

*killall(1M), rc2(1M), shutdown(1M).*

**NAME**

rc2 – run commands performed for multi-user environment

**SYNOPSIS**

**/etc/rc2**

**DESCRIPTION**

This file is executed via an entry in */etc/inittab* and is responsible for those initializations that bring the system to a ready-to-use state, traditionally state 2, called the "multi-user" state.

The actions performed by */etc/rc2* are found in files in the directory */etc/rc.d* and files beginning with *S* in */etc/rc2.d*. These files are executed by */bin/sh* in ascii sort–sequence order (see FILES for more information). When functions are added that need to be initialized when the system goes multi-user, an appropriate file should be added in */etc/rc2.d*.

The functions done by **/etc/rc2** command and associated */etc/rc2.d* files include:

Setting and exporting the TIMEZONE variable.

Setting-up and mounting the user (*/usr*) file system.

Cleaning up (remaking) the */tmp* and */usr/tmp* directories.

Loading the network interface and ports cards with program data and starting the associated processes.

Starting the *cron* daemon by executing */etc/cron*.

Cleaning up (deleting) uucp locks status, and temporary files in the */usr/spool/uucp* directory.

Other functions can be added, as required, to support the addition of hardware and software features.

**EXAMPLES**

The following are prototypical files found in */etc/rc2.d*. These files are prefixed by an **S** and a number indicating the execution order of the files.

MOUNTFILESYS

```
#   Set up and mount file systems

cd /
/etc/mountall /etc/fstab
```

RMTMPFILES

```
# clean up /tmp
rm -rf /tmp
mkdir /tmp
chmod 777 /tmp
chgrp sys /tmp
chown sys /tmp
```

uucp

```
#   clean-up uucp locks, status, and temporary files

rm -rf /usr/spool/locks/*
```

The file */etc/TIMEZONE* is included early in */etc/rc2*, thus establishing the default time zone for all commands that follow.

**FILES**

Here are some hints about files in */etc/rc.d*:

The order in which files are executed is important. Since they are executed in ascii sort-sequence order, using the first character of the file name as a sequence indicator will help keep the proper order. Thus, files starting with the following characters would be:

[0-9]. very early
[A-Z]. early
[a-n]. later
[o-z]. last

Files in */etc/rc.d* that begin with a dot (.) will not be executed. This feature can be used to hide files that are not to be executed for the time being without removing them. The command can be used only by the super-user.

Files in */etc/rc2.d* must begin with an **S** or a **K** followed by a number and the rest of the file name. Upon entering run level 2, files beginning with **S** are executed with the **start** option; files beginning with **K**, are executed with the **stop** option.. Files beginning with other characters are ignored.

**SEE ALSO**

*shutdown(1M)*.

**NAME**

rdump.ffs – file system dump across the network

**SYNOPSIS**

**/etc/rdump.ffs** [ key [ *argument* ... ] filesystem ]

**DESCRIPTION**

*rdump.ffs* copies to magnetic tape all files changed after a certain date in the *filesystem*. The command is identical in operation to *dump(1FFS)* except the *f* key should be specified and the file supplied should be of the form *machine:device*.

*rdump.ffs* creates a remote server, */etc/rmt*, on the client machine to access the tape device.

**SEE ALSO**

*dump(1FFS)*

**DIAGNOSTICS**

Same as *dump(1FFS)* with a few extra related to the network.

**NAME**

rdump - front-end for filesystem remote dump command

**SYNOPSIS**

**/etc/rdump** [ */etc/rdump.ffs arguments* ]

**DESCRIPTION**

This command is a front-end program that executes the command */etc/rdump.ffs* if all of the named filesystems (or */dev/root* by default) are ffs filesystems.

The key options **f**, **s**, and **d** are checked for corresponding arguments, and the **f** option *must* be given. If the **W** or **w** keys are given, no filesystem checking is done.

**SEE ALSO**

*rdump(1FFS).*

**NAME**

restore.ffs – incremental file system restore

**SYNOPSIS**

**/etc/restore.ffs** *key* [ *name* ... ]

**DESCRIPTION**

*restore.ffs* reads tapes dumped with the *dump(1FFS)* command. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying the files that are to be restored. Unless the **h** key is specified (see below), the appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

**r** The tape is read and loaded into the current directory. This should not be done lightly; the **r** key should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape after a full level zero restore. Thus

```
/etc/newfs.ffs /dev/rrp0g eagle
/etc/mount /dev/rp0g /mnt
cd /mnt
restore.ffs r
```

is a typical sequence to restore a complete dump. Another *restore.ffs* can be done to get an incremental dump in on top of this. Note that **restore.ffs** leaves a file *restoresymtab* in the root directory to pass information between incremental restore passes. This file should be removed when the last incremental tape has been restored.
A *dump(1FFS)* followed by a *newfs(1FFS)* and a *restore.ffs* is used to change the size of a file system.

**R** *restore.ffs* requests a particular tape of a multi volume set on which to restart a full restore (see the **r** key above). This allows **restore.ffs** to be interrupted and then restarted.

**x** The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, and the **h** key is not specified, the directory is recursively extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, then the root directory is extracted, which results in the entire content of the tape being extracted, unless the **h** key has been specified.

**t** The names of the specified files are listed if they occur on the tape. If no file argument is given, then the root directory is listed, which results in the entire content of the tape being listed, unless the **h** key has been specified. Note that the **t** key replaces the function of the old *dumpdir* program.

**i** This mode allows interactive restoration of files from a dump tape. After reading in the directory information from the tape, *restore.ffs* provides a shell like interface that allows the user to move around the directory tree selecting files to be extracted. The available commands are given below; for those commands that require an argument, the default is the current directory.

**ls** [arg] – List the current or specified directory. Entries that are directories are appended with a "/". Entries that have been marked for extraction are prepended with a "*". If the verbose key is set the inode number of each entry is also listed.

**cd** arg – Change the current working directory to the specified argument.

**pwd** – Print the full pathname of the current working directory.

**add** [arg] – The current directory or specified argument is added to the list of files to be extracted. If a directory is specified, then it and all its descendents are added to the extraction list (unless the **h** key is specified on the command line). Files that are on the extraction list are prepended with a "*" when they are listed by **ls**.

**delete** [arg] – The current directory or specified argument is deleted from the list of files to be extracted. If a directory is specified, then it and all its descendents are deleted from the extraction list (unless the **h** key is specified on the command line). The most expedient way to extract most of the files from a directory is to add the directory to the extraction list and then delete those files that are not needed.

**extract** – All the files that are on the extraction list are extracted from the dump tape. *restore.ffs* will ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

**setmodes** – All the directories that have been added to the extraction list have their owner, modes, and times set; nothing is extracted from the tape. This is useful for cleaning up after a restore has been prematurely aborted.

**verbose** – The sense of the **v** key is toggled. When set, the verbose key causes the **ls** command to list the inode numbers of all entries. It also causes **restore.ffs** to print out information about each file as it is extracted.

**help** – List a summary of the available commands.

**quit** – Restore immediately exits, even if the extraction list is not empty.

The following characters may be used in addition to the letter that selects the function desired.

**b** The next argument to **restore.ffs** is used as the block size of the tape (in kilobytes). If the **-b** option is not specified, **restore.ffs** tries to determine the tape block size dynamically.

**f** The next argument to **restore.ffs** is used as the name of the archive instead of /dev/rmt?. If the name of the file is "–", **restore.ffs** reads from standard input. Thus, *dump(1FFS)* and **restore.ffs** can be used in a pipeline to dump and restore a file system with the command

```
dump.ffs 0f - /usr | (cd /mnt; restore.ffs xf -)
```

**v** Normally **restore.ffs** does its work silently. The **v** (verbose) key causes it to type the name of each file it treats preceded by its file type.

**y** *restore.ffs* will not ask whether it should abort the restore.ffs if gets a tape error. It will always try to skip over the bad tape block(s) and

continue as best it can.

**m** *restore.ffs* will extract by inode numbers rather than by file name. This is useful if only a few files are being extracted, and one wants to avoid regenerating the complete pathname to the file.

**h** *restore.ffs* extracts the actual directory, rather than the files that it references. This prevents hierarchical restoration of complete subtrees from the tape.

**s** The next argument to **restore.ffs** is a number which selects the file on a multi-file dump tape. File numbering starts at 1.

**DIAGNOSTICS**

Complaints about bad key characters.

Complaints if it gets a read error. If **y** has been specified, or the user responds "y", **restore.ffs** will attempt to continue the restore.

If the dump extends over more than one tape, **restore.ffs** will ask the user to change tapes. If the **x** or **i** key has been specified, **restore.ffs** will also ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

There are numerous consistency checks that can be listed by **restore.ffs**. Most checks are self-explanatory or can "never happen". Common errors are given below.

Converting to new file system format.
A dump tape created from the old file system has been loaded. It is automatically converted to the new file system format.

<filename>: not found on tape
The specified file name was listed in the tape directory, but was not found on the tape. This is caused by tape read errors while looking for the file, and from using a dump tape created on an active file system.

expected next file <inumber>, got <inumber>
A file that was not listed in the directory showed up. This can occur when using a dump tape created on an active file system.

Incremental tape too low
When doing incremental restore, a tape that was written before the previous incremental tape, or that has too low an incremental level has been loaded.

Incremental tape too high
When doing incremental restore, a tape that does not begin its coverage where the previous incremental tape left off, or that has too high an incremental level has been loaded.

Tape read error while restoring <filename>
Tape read error while skipping over inode <inumber>
Tape read error while trying to resynchronize
A tape read error has occurred. If a file name is specified, then its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, then no extracted files have been corrupted, though files may not be found on the tape.

resync restore, skipped <num> blocks
After a tape read error, **restore.ffs** may have to resynchronize itself. This message lists the number of blocks that were skipped over.

**FILES**

| | |
|---|---|
| */dev/mt/ctape0* | the default tape drive |
| */tmp/rstdir** | file containing directories on the tape. |

*/tmp/rstmode** owner, mode, and time stamps for directories.
*/restoresymtable* information passed between incremental restores.

**SEE ALSO**

*dump(1FFS), newfs(1FFS), mount(1M), mkfs(1FFS)*

**ERRORS**

*restore.ffs* can get confused when doing incremental restores from dump tapes that were made on active file systems.

A level zero dump must be done after a full restore. Because *restore.ffs* runs in user code, it has no control over inode allocation; thus a full restore must be done to get a new set of directories reflecting the new inode numbering, even though the contents of the files is unchanged.

**NAME**

restore – front-end for filesystem restore command

**SYNOPSIS**

**/etc/restore** [ */etc/restore.ffs arguments* ]

**DESCRIPTION**

This command is a front-end program that executes the command */etc/restore.ffs*. This is done in the interest of completeness and to leave space for further development. No argument checking is done.

**SEE ALSO**

*restore(1FFS)*.

**NAME**

rexecd – remote execution server

**SYNOPSIS**

**/etc/rexecd**

**DESCRIPTION**

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

*rexecd* listens for service requests at the port indicated in the "exec" service specification; see *services(4)*. When a service request is received the following protocol is initiated:

1) The server reads characters from the socket 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, unencrypted 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 initial socket and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rexecd* .

**DIAGNOSTICS**

Except for the last one listed below, all diagnostic messages are returned on the initial socket, 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.

**"Password incorrect."**
The wrong was password supplied.

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

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

**"<shellname>: ..."**
The user's login shell could not be started. This message is returned on the connection associated with the *stderr*, and is not preceded by a flag byte.

**SEE ALSO**

*rexec(3X)*

**ERRORS**

Indicating "Login incorrect" as opposed to "Password incorrect" is a security breach which allows people to probe a system for users with null passwords.

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

**NAME**

rlogind – remote login server

**SYNOPSIS**

**/etc/rlogind** [ **–d** ]

**DESCRIPTION**

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

*rlogind* listens for service requests at the port indicated in the "login" service specification; see *services(4)*. 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 and requests the corresponding host name (see *gethostbyname(3N)* and *hosts(4)*). If the hostname cannot be determined, the dot-notation representation of the host address is used.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal (see *pty(7)*), 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 pseduo 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(7)* 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 environment variable, "TERM"; see *environ(5)*.

**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.

**"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.

A more extensible protocol should be used.

**ORIGIN**

4.3 BSD

NAME

route – manually manipulate the routing tables

SYNOPSIS

**/usr/etc/route** [ **–f** ] [ **–n** ] [ *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 two commands: *add*, to add a route, and *delete*, to delete a route.

All commands have the following syntax:

**/usr/etc/route** *command* [ **net** | **host** ] *destination gateway* [ *metric* ]

where *destination* is the destination host or network, *gateway* is the next-hop gateway to which packets should be addressed, and *metric* is a count indicating the number of hops to the *destination*. The metric is required for *add* commands; it must be zero if the destination is on a directly-attached network, and nonzero if the route utilizes one or more gateways. If adding a route with metric 0, the gateway given is the address of this host on the common network, indicating the interface to be used for transmission. Routes to a particular host are distinguished from those to a network by interpreting the Internet address associated with *destination*. The optional keywords **net** and **host** force the destination to be interpreted as a network or a host, respectively. Otherwise, if the *destination* has a "local address part" of INADDR_ANY, or if the *destination* is the symbolic name of a network, then the route is assumed to be 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 as a host name using *gethostbyname(3N)*. If this lookup fails, *getnetbyname(3N)* is then used to interpret the name as that of a network.

*route* uses a raw socket and the SIOCADDRT and SIOCDELRT *ioctl*'s to do its work. As such, only the super-user 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.

The **–n** option prevents attempts to print host and network names symbolically when reporting actions.

DIAGNOSTICS

"**add [ host | network ] %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. If the gateway address used was not the primary address of the gateway (the first one returned by *gethostbyname*), the gateway address is printed numerically as well as symbolically.

"**delete [ host | network ] %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.

"**Network is unreachable**"
An attempt to add a route failed because the gateway listed was not on a directly-connected network. The next-hop gateway must be given.

**"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(7P), routed(1M)*

**ORIGIN**

4.3 BSD

**NAME**

routed – network routing daemon

**SYNOPSIS**

**/etc/routed** [ **–d** ] [ **–g** ] [ **–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. It used a generalized protocol capable of use with multiple address types, but is currently used only for Internet routing within a cluster of networks.

In normal operation *routed* listens on the *udp*(4P) socket for the *route* service (see *services*(4)) 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 that 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 updates the kernel routing table. The change is reflected in the next *response* packet sent.

In addition to processing incoming packets, *routed* also periodically checks the routing table entries. If an entry has not been updated for 3 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 local internet.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks. The response is sent to the broadcast address on nets capable of that function, to the destination address on point-to-point links, and to the router's own address on other networks. The normal routing tables are bypassed when sending gratuitous responses. The reception of responses on each network is used to determine

that the network and interface are functioning correctly. If no response is received on an interface, another route may be chosen to route around the interface, or the route may be dropped if no alternative is available.

*routed supports several options:*

**-d** Enable additional debugging information to be logged, such as bad packets received.

**-g** This flag is used on internetwork routers to offer a route to the "default" destination. This is typically used on a gateway to the Internet, or on a gateway that uses another routing protocol whose routes are not reported to other local routers.

**-s** Supplying this option forces *routed* to supply routing information whether it is acting as an internetwork router or not. This is the default if multiple network interfaces are present, or if a point-to-point link is in use.

**-q** This is the opposite of the **-s** option.

**-t** 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, if not tracing all packets, 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 located using only information from 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. External gateways are also passive, but are not placed in the kernel routing table nor are they included in routing updates. The function of external entries is to inform *routed* that another routing process will install such a route, and that alternate routes to that destination should not be installed. Such entries are only required when both routers may learn of routes to the same destination.

The */etc/gateways* is comprised of a series of lines, each in the following format:

< **net** | **host** > *name1* **gateway** *name2* **metric** *value* < **passive** | **active** | **external** >

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, if started after *named(1M)*, known to the name server), 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.

One of the keywords **passive**, **active** or **external** indicates if the gateway should be treated as *passive* or *active* (as described above), or whether the gateway is external to the scope of the *routed* protocol.

Internetwork routers that are directly attached to the Arpanet or Milnet should use the Exterior Gateway Protocol (EGP) to gather routing information rather then using a static routing table of passive gateways. EGP is required in order to provide routes for local networks to the rest of the Internet system. Sites needing assistance with such configurations should contact the Computer Systems Research Group at Berkeley.

**FILES**

*/etc/gateways* for distant gateways

**SEE ALSO**

"Internet Transport Protocols", XSIS 028112, Xerox System Integration Standard.
*udp(4P), htable(1M)*

**ERRORS**

The kernel's routing tables may not correspond to those of *routed* when redirects change or add routes. The only remedy for this is to place the routing process in the kernel.

*routed* should incorporate other routing protocols, such as Xerox NS (*XNSrouted(8C)*) and EGP. Using separate processes for each requires configuration options to avoid redundant or competing routes.

*routed* should listen to intelligent interfaces, such as an IMP, and to error protocols, such as ICMP , to gather more information. It does not always detect unidirectional failures in network interfaces (e.g., when the output side fails).

**ORIGIN**

U.C. Berkeley, with changes from MIPS Computer Systems, Inc.

**NAME**

rpc.passwd – server for modifying password file

**SYNOPSIS**

**/usr/etc/rpc.passwd** *file* [ **–m** *arg1 arg2 ...* ]

**DESCRIPTION**

*rpc.passwd* is a server that responds to remote password protocol. It changes a password entry in *file*, which is assumed to be in the format of *passwd(4)*. An entry in *file* will only be changed if the password presented matches the encrypted password of that entry.

If the **–m** option is given, then after *file* is modified, a *make(1)* will be performed in */usr/etc/yp*. Any arguments following the flag will be passed to *make*.

This server is not run by default, nor can it be started up from *inetd(1M)*.

**NOTES**

*/etc/rpc.passwd* is a symbolic link to */usr/etc/spc.passwd*.

**FILES**

*/usr/etc/yp/Makefile*

**SEE ALSO**

*passwd(4)* in the *Programmer's Reference Manual.*

**CAVEAT**

This server will eventually be replaced with a more general service.

**ORIGIN**

Sun Microsystems

**NAME**

rpcinfo – report RPC information

**SYNOPSIS**

**rpcinfo −p** [ *host* ]
**rpcinfo −u** *host program-number* [ *version-number* ]
**rpcinfo −t** *host program-number* [ *version-number* ]

**DESCRIPTION**

*rpcinfo* makes an RPC call to an RPC server and reports what it finds.

**OPTIONS**

**−p** Probe the portmapper on *host*, and print a list of all registered RPC programs. If *host* is not specified, it defaults to the value returned by *hostname(1)*.

**−u** Make an RPC call to procedure 0 of *program-number* using UDP, and report whether a response was received.

**−t** Make an RPC call to procedure 0 of *program-number* using TCP, and report whether a response was received.

The *program-number* argument can be either a name or a number. If no version is given, it defaults to 1.

**FILES**

*/etc/rpc* names for rpc program numbers

**SEE ALSO**

*RPC Programming Guide, rpc(4), portmap(1m)*

**ORIGIN**

Sun Microsystems

**NAME**

rrestore.ffs – restore a file system dump across the network

**SYNOPSIS**

**/etc/rrestore.ffs** [ *key* [ *name* ... ] ]

**DESCRIPTION**

*rrestore.ffs* obtains, from magnetic tape, files saved by a previous *dump(1FFS)*. The command is identical in operation to *restore(1FFS)* except the *f* key should be specified and the file supplied should be of the form *machine:device*.

*rrestore.ffs* creates a remote server, */etc/rmt*, on the client machine to access the tape device.

**SEE ALSO**

*restore(1FFS)*

**DIAGNOSTICS**

Same as *restore(1FFS)* with a few extra related to the network.

**NAME**

rrestore – front-end for filesystem remote rrestore command

**SYNOPSIS**

**/etc/rrestore** [ */etc/rrestore.ffs arguments* ]

**DESCRIPTION**

This command is a front-end program that executes the command */etc/rrestore.ffs*. This is done in the interest of completeness and to leave space for further development. No argument checking is done.

**SEE ALSO**

*rrestore(1FFS)*.

**NAME**

rshd – remote shell server

**SYNOPSIS**

**/etc/rshd**

**DESCRIPTION**

*rshd* is the server for the *rcmd(3X)* routine and, consequently, for the *rsh(1C)* program. The server provides remote execution facilities with authentication based on privileged port numbers from trusted hosts.

*rshd* listens for service requests at the port indicated in the "cmd" service specification; see *services(4)*. 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 and requests the corresponding host name (see *gethostbyaddr(3N), hosts(4)* and *named(1M)*). If the hostname cannot be determined, the dot-notation representation of the host address is used.

5) 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.

6) 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.

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 local (server-end) 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 super-user, (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 super-user, 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 initial socket and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by **rshd.**

**DIAGNOSTICS**

Except for the last one listed below, all diagnostic messages are returned on the initial socket, 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 execution of the login shell).

**"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).

**"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.

**"<shellname>: ..."**
The user's login shell could not be started. This message is returned on the connection associated with the *stderr,* and is not preceded by a flag byte.

**SEE ALSO**

*rsh(1C), rcmd(3X)*

**ERRORS**

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.

A more extensible protocol should be used.

**NAME**

rwalld – network rwall server

**SYNOPSIS**

**/usr/etc/rpc.rwalld**

**DESCRIPTION**

*rwalld* is a server that handles *rwall(1)* and *shutdown(1)* requests. It is implemented by calling *wall(1)* to all the appropriate network machines. The *rwalld* daemon is normally invoked by *inetd(1M)*.

**SEE ALSO**

*rwall(1), wall(1), inetd(1M)*

**ORIGIN**

Sun Microsystems

**NAME**

rwhod – system status server

**SYNOPSIS**

**/etc/rwhod**

**DESCRIPTION**

*rwhod* is the server which maintains the database used by the *rwho(1C)* and *ruptime(1C)* 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 server transmits and receives messages at the port indicated in the "rwho" service specification; see *services(4)*. 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 as calculated by the *uptime(1)* program, and represent load averages over the 5, 10, and 15 minute intervals prior to a server's transmission; they are multiplied by 100 for representation in an integer.

The host name included is that returned by the *gethostname(2)* system call, with any trailing domain name omitted. The array at the end of the message contains information about the users logged in to the sending machine. This information includes the contents of the *utmp(4)* entry for each non-idle terminal line and a value indicating the time in seconds 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 3 minutes. *rwhod* performs an *nlist(3)* on /vmunix every 30 minutes to guard against the possibility that this file is not the system image currently operating.

**NOTES**

*/etc/rwhod* is a symbolic link to */usr/etc/rwho*

**SEE ALSO**

*rwho(1C), ruptime(1C)*

**ERRORS**

There should be a way to relay status information between networks. Status information should be sent only upon request rather than continuously. People often interpret the server dying or network communtication failures as a machine going down.

**NAME**

sar: sa1, sa2, sadc – system activity report package

**SYNOPSIS**

**/usr/lib/sa/sadc** [*t n*] [ *ofile* ]

**/usr/lib/sa/sa1** [*t n*]

**/usr/lib/sa/sa2** [**–ubdycwaqvmprSDA**] [**–s** *time* ] [**–e** *time* ] [**–i** *sec* ]

**DESCRIPTION**

System activity data can be accessed at the special request of a user (see *sar(1)*) and automatically on a routine basis as described here. The operating system contains a number of counters that are incremented as various system actions occur. These include counters for CPU utilization, buffer usage, disk and tape I/O activity, TTY device activity, switching and system-call activity, file-access, queue activity, inter-process communications, paging and Remote File Sharing.

*sadc* and shell procedures, *sa1* and *sa2*, are used to sample, save, and process this data.

*sadc*, the data collector, samples system data *n* times every *t* seconds and writes in binary format to *ofile* or to standard output. If *t* and *n* are omitted, a special record is written. This facility is used at system boot time, when booting to a multiuser state, to mark the time at which the counters restart from zero. For example, the */etc/init.d/perf* file writes the restart mark to the daily data by the command entry:

```
su sys -c "/usr/lib/sa/sadc /usr/adm/sa/sa`date +%d`"
```

The shell script *sa1*, a variant of *sadc*, is used to collect and store data in binary file */usr/adm/sa/sa***dd** where **dd** is the current day. The arguments *t* and *n* cause records to be written *n* times at an interval of *t* seconds, or once if omitted. The entries in */usr/spool/cron/crontabs/sys* (see *cron(1M)*):

```
0 * * * 0-6 /usr/lib/sa/sa1
20,40 8-17 * * 1-5 /usr/lib/sa/sa1
```

will produce records every 20 minutes during working hours and hourly otherwise.

The shell script *sa2*, a variant of *sar(1)*, writes a daily report in file */usr/adm/sa/sar***dd**. The options are explained in *sar(1)*. The */usr/spool/cron/crontabs/sys* entry:

```
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

will report important activities hourly during the working day.

The structure of the binary daily data file is:

```
struct sa {
        struct sysinfo si;  /* see /usr/include/sys/sysinfo.h */
        struct minfo mi;  /* defined in sys/sysinfo.h */
        struck dinfo di;  /* RFS info defined in sys/sysinfo.h */
        int minserve, maxserve;    /* RFS server low and high water marks */
        int  szinode;       /* current size of inode table  */
        int  szfile;        /* current size of file table  */
        int  szproc;        /* current size of proc table  */
        int   szlckf;       /* current size of file record header table */
        int   szlckr;       /* current size of file record lock table */
        int  mszinode;      /* size of inode table  */
        int  mszfile;       /* size of file table  */
        int  mszproc;       /* size of proc table  */
        int   mszlckf;      /* maximum size of file record header table */
        int   mszlckr;      /* maximum size of file record lock table */
        long  inodeovf;     /* cumulative overflows of inode table  */
        long  fileovf;      /* cumulative overflows of file table  */
        long  procovf;      /* cumulative overflows of proc table  */
        time_t  ts;         /* time stamp, seconds  */
        long  devio[NDEVS][4];    /* device unit information  */
#define IO_OPS          0       /* cumulative I/O requests  */
#define IO_BCNT         1       /* cumulative blocks transferred */
#define IO_ACT          2       /* cumulative drive busy time in ticks  */
#define IO_RESP         3       /* cumulative I/O resp time in ticks */
};
```

**FILES**

| | |
|---|---|
| */usr/adm/sa/sa***dd** | daily data file |
| */usr/adm/sa/sar***dd** | daily report file |
| */tmp/sa.adrfl* | address file |

**SEE ALSO**

*cron(1M).*

*sag(1G), sar(1), timex(1)* in the *User's Reference Manual.*

NAME

sash – general description of the standalone shell

SYNOPSIS

**sash** [ **-a** ] [ **-r** ] [ file [ args ]]

DESCRIPTION

*Sash* is the MIPS standalone shell. The standalone shell is an extended version of the PROM Monitor that includes all the PROM Monitor commands. In addition to the PROM Monitor commands, sash includes additional commands and is configured with more device drivers and file system types. Sash exists so that the MIPS standalone programs and the PROM Monitor are not dependent on the operating system.

The sash program is booted using the PROM Monitor Boot command. The sash program can be booted from a cartridge tape, from a hard disk if the software has already been installed, or from the network. To boot the sash program from the network, a machine must be running the bootfile Server Daemon **bfsd(8).**

To load the sash program from the network, type:

boot -f bfs()sash [**-a**][**-r**] [file[args]]

The parenthesis in the commands shown above indicate that the previous argument is a device. When booting over the network, if the command is entered as shown, then it will boot sash from the first machine that has the program the machine name and path name.

If sash is booted without arguments, then the sash command mode is entered. The sash command prompts is shown below.

sash:

If the **-a** argument is used as the first argument, then sash assumes that an *automatic* operating system boot is to be done. Sash examines the name by which it was booted and uses the same device, controller, and unit to look for an operating system to boot. Sash finds the correct operating system file to boot by examining the disk volume header on the specified device. The volume header specifies a root partition and an operating system file name. Once the appropriate operating system file is determined, sash boots the operating system and passes the **-a** argument and any other arguments following the **-a** to the operating system.

If the -r argument is specified as the first argument, then sash assumes that the next argument is a standalone program that is being booted by a remote debugger. Sash defines the environment variables "dbgmon" and "rdebug", boots the file specified by the argument after the **-r** flag, and passes any succeeding arguments. If the booted program was linked against the standalone library, then the start-up code provided will note the environment variables "dbgmon" and "rdebug" and load the debugging monitor co-resident with the program. This causes the program to enter the remote debugging mode.

If any other argument is passed to sash when it is booted, then sash interprets the argument as the file name of a program to be booted immediately. Any other arguments appearing on the command line to call sash will be passed through to the booted program. Therefore, if the PROM Monitor environment variable **bootfile** is set as "sash" and the command listed below was entered on the PROM Monitor command line, then the PROM Monitor would load the file indicated by the environment variable **bootfile.** The bootfile contains the sash program.

boot dkis()unix or boot dkip()unix or boot dksd()unix

EXTENDING THE STANDALONE SHELL

If you type a sash command on the sash command line that is not built-in , then sash uses the first word of the command as the name of a file. Sash then tries to boot that file by passing

any other arguments on the command line to the booted program. This mechanism makes two-level boots possible.

If the environment variable **$path** is not defined, then the first word of the command must be a complete file name specification consisting of a device name, controller, unit, partition, and a file path. If the environment variable **$path** formed by prepending the contents of **$path** to the original file name. If **$path** is a list of prefixes separated by spaces, then the standalone shell will try each prefix from **$path** until the file is successfully booted or until all prefixes have been tried.

## SASH COMMANDS

When sash is booted without arguments, the sash command mode is entered. From the command mode prompt, memory and environment variables can be displayed and altered, and other programs can be booted. All of the commands shown below except **cp(1)** are PROM Monitor commands and can be found in their own man page listing.

| | |
|---|---|
| auto | Initiates the two-level operating system autoboot sequence. |
| boot | Loads the specified program. |
| cat | Displays the contents of the files listed on the console. |
| cp | Copies the contents of one file to another file. |
| disable | Does not allow input from an output to the specified console device. |
| dump | Formats and displays the contents of memory. |
| enable | Allows input from and output to the specified console device. |
| fill | Fills the specified range of memory with the specified pattern. |
| g | Displays the contents of a single memory location in decimal, hexadecimal, and ASCII character formats. |
| go | Transfers control to code that is assumed to have been previously loaded. |
| help | Displays the syntax for all commands. |
| init | Reinitializes the PROM Monitor software state. |
| init_tod | Initializes the time-of-day chip. |
| inittod | Initializes M/2030 time-of-day chip. |
| load | Allows you to load memory over a serial line connection from a system running the RISC/os program **cu(1).** |
| p | Puts or sets the contents of a single memory location to a specified value. |
| printenv | Displays the value of the PROM environment variables. |
| setenv | Used to create a new environment variable or to change an existing environment variable. |
| sload | Accepts a subset of the Motorola S-record protocol. |
| spin | Generates reference patterns for diagnostic use. |
| unsetenv | Used to delete an existing environment variable. |
| warm | Examines memory for a restart block. |

## SEE ALSO

intro(1spp), prom(1Mspp), dbgmon(1spp)

**NAME**

savecore – save a core dump of the operating system

**SYNOPSIS**

**/etc/savecore** *dirname* [ *system* ]

**DESCRIPTION**

*savecore* is meant to be called near the end of the system initialization process. Its function is to save the core dump of the system (assuming one was made) and to write a reboot message in the shutdown log.

Savecore checks the core dump to be certain it corresponds with the current running unix. If it does it saves the core image in the file *dirname*/core.n and its brother, the namelist, *dirname*/unix.n The trailing ".n" in the pathnames is replaced by a number which grows every time *savecore* is run in that directory.

Before savecore writes out a core image, it reads a number from the file *dirname*/minfree. If the number of free kilobytes on the filesystem which contains *dirname* is less than the number obtained from the minfree file, the core dump is not saved. If the minfree file does not exist, savecore always writes out the core file (assuming that a core dump was taken).

*savecore* also logs a reboot message using facility LOG_AUTH (see *syslog*(3)) If the system crashed as a result of a panic, *savecore* logs the panic string too.

If the core dump was from a system other than /unix, the name of that system must be supplied as *sysname*.

**FILES**

/unix current UNIX

**ERRORS**

Can be fooled into thinking a core dump is the wrong size.

**NAME**

sccstorcs – build RCS file from SCCS file.

**SYNOPSIS**

**sccstorcs** [**–t**] [**–v**] *s.file* ...

**DESCRIPTION**

*sccstorcs* builds an RCS file from each SCCS file argument. The deltas and comments for each delta are preserved and installed into the new RCS file in order. Also preserved are the user access list and descriptive text, if any, from the SCCS file.

The following flags are meaningful:

**–t** Trace only. Prints detailed information about the SCCS file and lists the commands that would be executed to produce the RCS file. No commands are actually executed and no RCS file is made.

**–v** Verbose. Prints each command that is run while it is building the RCS file.

**FILES**

For each s.*somefile*, **sccstorcs** writes the files *somefile* and *somefile*,v which should not already exist. *sccstorcs* will abort, rather than overwrite those files if they do exist.

**SEE ALSO**

*ci (1), co (1), rcs (1).*

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

**DIAGNOSTICS**

All diagnostics are written to *stderr*. Non-zero exit status on error.

**ERRORS**

*sccstorcs* does not preserve all SCCS options specified in the SCCS file. Most notably, it does not preserve removed deltas, MR numbers, and cutoff points.

**AUTHOR**

Ken Greer

**NAME**

sendmail – send mail over the internet

**SYNOPSIS**

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

**newaliases**

**mailq** [ **–v** ]

**DESCRIPTION**

*sendmail* sends a message to one or more *recipients*, 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 an end-of-file or a line consisting only of a single dot and sends a copy of the message found there to all of the addresses listed. It determines the network(s) 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. This requires Berkeley IPC. *sendmail* will fork and run in background listening on socket 25 for incoming SMTP connections. This is normally run from */etc/rc*.

**–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 on standard input and output. 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. Verify mode is normally used for validating users or mailing lists.

**–bz** Create the configuration freeze file.

**–C***file* Use alternate configuration file. *sendmail* refuses to run as root if an alternate configuration file is specified. The frozen configuration file is bypassed.

**–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 "trusted" users (normally *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. If not specified, "Received:" lines in the message are counted.

**−n** Don't do aliasing.

**−o***x value* Set option *x* to the specified *value*. Options are described below.

**−q**[*time*] Processed 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. If *time* is specified, *sendmail* will run in background. This option can be used safely with **−bd**.

**−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 recipient addresses. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed, that is, they will *not* receive copies even if listed in the message header.

**−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. These are described in detail in the *Sendmail Installation and Operation Guide*. The options are:

A*file* 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), 'q' to throw away error messages (only exit status is returned), and 'e' to do special processing for the BerkNet. 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. This option violates the word (if not the intent) of the SMTP specification, show the timeout should probably be fairly large. |
| S*file* | Save statistics in the named file. |
| s | Always instantiate the queue file, even under circumstances where it is not strictly necessary. This provides safety against system crashes during delivery. |
| T*time* | Set the timeout on undelivered messages in the queue to the specified time. After delivery has failed (e.g., because of a host being down) for this amount of time, failed messages 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. |

In aliases, the first character of a name may be a vertical bar to cause interpretation of the rest of the name as a command to pipe the mail to. It may be necessary to quote the name to keep *sendmail* from suppressing the blanks from between arguments. For example, a common alias is:

```
msgs: "|/usr/ucb/msgs -s"
```

Aliases may also have the syntax ":include:*filename*" to ask *sendmail* to read the named file for a list of recipients. For example, an alias such as:

```
poets: ":include:/usr/local/lib/poets.list"
```

would read */usr/local/lib/poets.list* for the list of addresses making up the group.

*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/spool/mqueue/* | temp files |

**SEE ALSO**

*mail(1), rmail(1), aliases(4), forward(4), mailaddr(7);*
DARPA Internet Request For Comments RFC819, RFC821, RFC822;

**NAME**

setenv – set prom environment variable

**SYNOPSIS**

**setenv** var value

**DESCRIPTION**

The *setenv* command is used to create a new environment variable or to change the value of an existing environment variable. Environment variables are represented as ASCII strings. The current value of environment variables are passed to programs booted by the PROM Monitor or the standalone shell ( *sash* ). Keep in mind that changing the environment variables passed by the PROMS does not change the values in NURAM.

**SEE ALSO**

printenv(1prom), unsetenv(1prom), intro(1spp)

**NAME**

setmnt – establish mount table

**SYNOPSIS**

**/etc/setmnt**

**DESCRIPTION**

*setmnt* creates the */etc/mnttab* table which is needed for both the *mount(1M)* and *umount* commands. *setmnt* reads standard input and creates a *mnttab* entry for each line. Input lines have the format:

filesys node

where *filesys* is the name of the file system's *special file* (e.g., */dev/dsk/c?d?s?*) and *node* is the root name of that file system. Thus *filesys* and *node* become the first two strings in the mount table entry.

**FILES**

*/etc/mnttab*

**SEE ALSO**

*mount(1M)*.

**ERRORS**

Problems may occur if *filesys* or *node* are longer than 32 characters.
*setmnt* silently enforces an upper limit on the maximum number of *mnttab* entries.

**NAME**

showmount – show all remote mounts

**SYNOPSIS**

**/usr/etc/showmount** [ **–a** ] [ **–d** ] [ **–e** ] [ *host* ]

**DESCRIPTION**

*showmount* lists all the clients that have remotely mounted a filesystem from *host*. This information is maintained by the *mountd(1M)* server on *host*, and is saved across crashes in the file */etc/rmtab*. The default value for *host* is the value returned by *hostname(1)*.

**OPTIONS**

**–d** List directories that have been remotely mounted by clients.

**–a** Print all remote mounts in the format

hostname:directory

where *hostname* is the name of the client, and *directory* is the root of the file system that has been mounted.

**–e** Print the list of exported file systems.

**SEE ALSO**

*mountd(1M), exports(4)*

**ERRORS**

If a client crashes, its entry will not be removed from the list until it reboots and executes *umount –a*.

**ORIGIN**

Sun Microsystems

**NAME**

shutdown – shut down system, change system state

**SYNOPSIS**

**/etc/shutdown** [ **–y** ] [ **–g** *grace_period* [ **–i** *init_state* ]

**DESCRIPTION**

This command is executed by the super-user to change the state of the machine. By default, it brings the system to a state where only the console has access to the UNIX system. This state is traditionally called "single-user".

The command sends a warning message and a final message before it starts actual shutdown activities. By default, the command asks for confirmation before it starts shutting down daemons and killing processes. The options are used as follows:

**–y** pre-answers the confirmation question so the command can be run without user intervention. A default of 60 seconds is allowed between the warning message and the final message. Another 60 seconds is allowed between the final message and the confirmation.

**–g** *grace_period* allows the super-user to change the number of seconds from the 60-second default.

**–i** *init_state* specifies the state that *init(1M)* is to be put in following the warnings, if any. By default, system state "**s**" is used (the same as states "**1**" and "**S**").

Other recommended system state definitions are:

state 0 Shut the machine down so it is safe to remove the power. Have the machine remove power if it can. The */etc/rc0* procedure is called to do this work.

state 1, s, S
Bring the machine to the state traditionally called single-user. The */etc/rc0* procedure is called to do this work. (Though **s** and **1** are both used to go to single user state, **s** only kills processes spawned by **init** and does not unmount file systems. State **1** unmounts everything except root and kills all user processes, except those that relate to the console.)

state 5 Stop the UNIX system and go to the firmware monitor.

state 6 Stop the UNIX system and reboot to the state defined by the *initdefault* entry in */etc/inittab*.

**SEE ALSO**

*init(1M), rc0(1M), rc2(1M).*
*inittab(4)* in the *Programmer's Reference Manual.*

NAME

sload – download Motorola S-record images via a serial line

SYNOPSIS

**sload** [**-a**] console_device

DESCRIPTION

*sload* accepts a subset of the Motorola S-record protocol. The record types accepted are: 0, 3, and 7. You can use the System Programmer's Package commands *convert (1spp)* to produce S-record images and *sdownload (1spp)* to download the S-record images.

If you do not specify a **-a** , then the PROM replies with an ASCII ACK to each S-record received that has a valid checksum. The PROM replies with an ASCII NACK for records that have incorrect checksums.

BUGS

*sload* has not been debugged. Consider it a starting point.

SEE ALSO

convert(1spp), sdownload(1spp), load(1prom), srec(5spp)

**NAME**

spin – diagnostic reference pattern generator

**SYNOPSIS**

**spin** [[ -c count ] [ -v value ] -(r|w)(b|h|w) address ]+ [ -c count ]

**DESCRIPTION**

*spin* generates reference patterns for diagnostic use. You can specify a sequence of reads and/or writes of byte (b), halfword (h), or word (w) width with combinations of **-r** and **-w** options. The **-c** option specifies a repetition count and a value for writes that apply to all succeeding writes. The **-v** option specifies a count and a value for writes that apply for all succeeding reads and writes. A final count specification indicates the number of times the entire preceding pattern should be repeated. Count defaults to 1 and value defaults to 0. A negative count is interpreted as infinity.

**EXAMPLE**

>> spin -c 4 -rh 0x2 -c 2 -v 1 -wb 0x4 -c 10

The example shown above, repeats the two instructions shown below, 10 times.

Read the halfword at address 0x2 four times
Write 1 to the byte at address 0x4 two times

**NAME**

spray – spray packets

**SYNOPSIS**

**/usr/etc/spray** *host* [ **–l** ] [ **–nth** ] [ **–c –cnt** ]

**DESCRIPTION**

*spray* sends a one-way stream of packets to *host* using rpc, and then reports how many were received by *host* and what the transfer rate was. The default value of *lnth* is 86 bytes (the size of the rpc and udp headers), and the value of *cnt* is the number of packets required to make the total stream size 10000 bytes. The host name can be either a name or an internet address.

The *lnth* parameter is the numbers of bytes in the ethernet packet that holds the rpc call message. Since the data is encoded using xdr, and xdr only deals with 32 bit quantities, not all values of *lnth* are possible. *spray* will round up to the nearest possible value. When *lnth* is greater than 1514, then the rpc call can no longer be encapsulated in one ethernet packet, so the *lnth* field no longer has a simple correspondence to ethernet packet size.

**ORIGIN**

Sun Microsystems

**NAME**

stamp_links – setup compiler/include/library links for a given version stamp

**SYNOPSIS**

**/etc/stamp_links** [ **–f** ] *stamp*

**DESCRIPTION**

*stamp_links* sets up symbolic links in /bin, /lib, /usr/bin, /usr/lib, /usr/new, /usr/new/lib, and /usr, so that any command, directory, or data file whose name ends with *stamp* is pointed to by a symbolic link without the stamp. This is usually used to change over to a different version of the compiler system, but may have other applications as well.

The directory /usr/include is treated specially: all files found in the stamped include directory are copied into /usr/include.

The *stamp* argument must be of the form *digit.digit* unless the **–f** option is given, in which case no checking is done with respect to the stamp contents.

For example, if you are using compiler version 1.0, have recieved and installed version 1.1, and wish to make the default version 1.1, you would execute the command:

```
stamp_links 1.1
```

**SEE ALSO**

ln(1)

**ERRORS**

Between compiler versions 1.1 and 1.11, there were some organizational changes, so that you can upgrade from an older compiler system to 1.11, but going the other way can not be done automatically.

**NAME**

strace – print STREAMS trace messages

**SYNOPSIS**

**strace** [ *mid sid level* ] ...

**DESCRIPTION**

*strace* without arguments writes all STREAMS event trace messages from all drivers and modules to its standard output. These messages are obtained from the STREAMS log driver [*log(7)*]. If arguments are provided they must be in triplets of the form *mid, sid, level*, where *mid* is a STREAMS module id number, *sid* is a sub-id number, and *level* is a tracing priority level. Each triplet indicates that tracing messages are to be received from the given module/driver, sub-id (usually indicating minor device), and priority level equal to or less than the given level. The token *all* may be used for any member to indicate no restriction for that attribute. The format of each trace message output is: <seq> <time> <ticks> <level> <flags> <mid> <sid> <text>

| | |
|---|---|
| <seq> | trace sequence number |
| <time> | time of message in hh:mm:ss |
| <ticks> | time of message in machine ticks since boot |
| <level> | tracing priority level |
| <flags> | E : message is also in the error log<br>F : indicates a fatal error<br>N : mail was sent to the system administrator |
| <mid> | module id number of source |
| <sid> | sub-id number of source |
| <text> | formatted text of the trace message Once initiated, **strace** will continue to execute until terminated by the user. |

**EXAMPLES**

Output all trace messages from the module or driver whose module id is 41:

strace 41 all all Output those trace messages from driver/module id 41 with sub-ids 0, 1, or 2:

strace 41 0 1 41 1 1 41 2 0 Messages from sub-ids 0 and 1 must have a tracing level less than or equal to 1. Those from sub-id 2 must have a tracing level of 0.

**CAVEATS**

Due to performance considerations, only one **strace** process is permitted to open the STREAMS log driver at a time. The log driver has a list of the triplets specified in the command invocation, and compares each potential trace message against this list to decide if it should be formatted and sent up to the **strace** process. Hence, long lists of triplets will have a greater impact on overall STREAMS performance. Running **strace** will have the most impact on the timing of the modules and drivers generating the trace messages that are sent to the **strace** process. If trace messages are generated faster than the **strace** process can handle them, then some of the messages will be lost. This last case can be determined by examining the sequence numbers on the trace messages output.

**SEE ALSO**

*log(7).*
*STREAMS Programmer's Guide.*

**NAME**

strclean – STREAMS error logger cleanup program

**SYNOPSIS**

**strclean** [ **-d** *logdir* ] [**-a** *age* ]

**DESCRIPTION**

*strclean* is used to clean up the STREAMS error logger directory on a regular basis (for example, by using *cron(1M)*). By default, all files with names matching *error.** in */usr/adm/streams* that have not been modified in the last 3 days are removed. A directory other than */usr/adm/streams* can be specified using the **-d** option. The maximum age in days for a log file can be changed using the **-a** option.

**EXAMPLE**

strclean -d /usr/adm/streams -a 3 has the same result as running **strclean** with no arguments.

**NOTES**

*strclean* is typically run from *cron(1M)* on a daily or weekly basis.

**FILES**

*/usr/adm/streams/error.**

**SEE ALSO**

*cron(1M), strerr(1M).*
*STREAMS Programmer's Guide.*

**NAME**

strerr – STREAMS error logger daemon

**SYNOPSIS**

**strerr**

**DESCRIPTION**

*strerr* receives error log messages from the STREAMS log driver [*log(7)*] and appends them to a log file. The error log files produced reside in the directory */usr/adm/streams*, and are named *error.***mm-dd** where *mm* is the month and *dd* is the day of the messages contained in each log file. The format of an error log message is: <seq> <time> <ticks> <flags> <mid> <sid> <text>

| | |
|---|---|
| <seq> | error sequence number |
| <time> | time of message in hh:mm:ss |
| <ticks> | time of message in machine ticks since boot priority level |
| <flags> | T : the message was also sent to a tracing process<br>F : indicates a fatal error<br>N : send mail to the system administrator |
| <mid> | module id number of source |
| <sid> | sub-id number of source |
| <text> | formatted text of the error message Messages that appear in the error log are intended to report exceptional conditions that require the attention of the system administrator. Those messages which indicate the total failure of a STREAMS driver or module should have the F flag set. Those messages requiring the immediate attention of the administrator will have the N flag set, which causes the error logger to send the message to the system administrator via *mail(1)*. The priority level usually has no meaning in the error log but will have meaning if the message is also sent to a tracer process. |

Once initiated, **strerr** will continue to execute until terminated by the user. Commonly, **strerr** would be executed asynchronously.

**CAVEATS**

Only one **strerr** process at a time is permitted to open the STREAMS log driver.

If a module or driver is generating a large number of error messages, running the error logger will cause a degradation in STREAMS performance. If a large burst of messages are generated in a short time, the log driver may not be able to deliver some of the messages. This situation is indicated by gaps in the sequence numbering of the messages in the log files.

**FILES**

*/usr/adm/streams/error.***mm-dd**

**SEE ALSO**

*log(7).*
*STREAMS Programmer's Guide.*

**NAME**

**su, ssu** - substitute user id temporarily

**SYNOPSIS**

**su** [ **−f** ] [ **−** ] [ **−e** ] [ **−c** ] [ *userid* [ *command* [ *args...* ] ] ]

**DESCRIPTION**

*su* demands the password of the specified *userid,* and if it is given, changes to that *userid* and invokes the shell (unless **−c** is given, see below) without changing the current directory. The user environment is unchanged. (see *environ(7)*). The new user ID stays in force until the shell exits.

If no *userid* is specified, "root" is assumed. Only users in the "wheel" group (group 0) or in the file */etc/su_people* (described below) can *su* to "root", even with the root password (this can be overridden by changing *su* to have group wheel and turning on the set-group-id permission). To remind the super-user of his responsibilities, the shell substitutes '#' for its usual prompt.

The command *ssu* is a link to *su*. Executing *ssu* is the same as executing the command 'su -c root'.

If the user tries to **su** to "root" and the root account has a password (as is the preferable case), the file */etc/su_people* is read to see if that username is allowed to become root without a password. Since this can be dangerous, the file must have owner 0 (root), group root (0), and mode 0600 (read and write by owner only), or it will be silently ignored. See the manual page for *su_people(4)* for details on this file.

**OPTIONS**

| | |
|---|---|
| **−f** | Prevents *csh(1)* from executing the *.cshrc* file; thus making *su* start up faster. |
| **−** | Simulates a full login by executing the shell with name '-sh'. |
| **−e** | This option has no effect, and is provided for compatability with the UMIPS-BSD *su* command. |
| **−c** | If any arguments are given after the username, they are executed as a command instead of the shell. For example, 'su -c root ls' will execute the command *ls(1)* as root, whereas 'su root ls' will execute the command 'csh ls' as root (this is not the same thing). |

**FILES**

| | |
|---|---|
| */etc/su_people* | Special permission database |

**SEE ALSO**

*sh(1), csh(1), su_people(4)*

**ERRORS**

The default semantics for *su* changed in System V Release 3.0, and there is no way to obtain the results of previous versions.

NAME

swap – swap administrative interface

SYNOPSIS

**/etc/swap −a** *swapdev swaplow swaplen*
**/etc/swap −d** *swapdev swaplow*
**/etc/swap −l**

DESCRIPTION

*swap* provides a method of adding, deleting, and monitoring the system swap areas used by the memory manager. The following options are recognized:

**−a** Add the specified swap area. *swapdev* is the name of the block special device, e.g., */dev/dsk/1s0*. *swaplow* is the offset in 512-byte blocks into the device where the swap area should begin. *swaplen* is the length of the swap area in 512-byte blocks. This option can only be used by the super-user. Swap areas are normally added by the system start up routine */etc/rc* when going into multi-user mode.

**−d** Delete the specified swap area. *swapdev* is the name of block special device, e.g., */dev/dsk/1s0*. *swaplow* is the offset in 512-byte blocks into the device where the swap area should begin. Using this option marks the swap area as "INDEL" (in process of being deleted). The system will not allocate any new blocks from the area, and will try to free swap blocks from it. The area will remain in use until all blocks from it are freed. This option can only be used by the super-user.

**−l** List the status of all the swap areas. The output has four columns:

**DEV** The *swapdev* special file for the swap area if one can be found in the */dev/dsk* or */dev* directories, and its major/minor device number in decimal.

**LOW** The *swaplow* value for the area in 512-byte blocks.

**LEN** The *swaplen* value for the area in 512-byte blocks.

**FREE** The number of free 512-byte blocks in the area. If the swap area is being deleted, this column will be marked INDEL.

WARNINGS

No check is done to see if a swap area being added overlaps with an existing swap area or file system.

**NAME**

sync – update the super block

**SYNOPSIS**

*sync*

**DESCRIPTION**

*sync* executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to insure file system integrity. It will flush all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point will be saved. See *sync(2)* for details.

**NOTE**

If you have done a write to a file on a remote machine in a Remote File Sharing environment, you cannot use *sync* to force buffers to be written out to disk on the remote machine. *sync* will only write local buffers to local disks.

**SEE ALSO**

*sync(2)* in the *Programmer's Reference Manual.*

**NAME**

telnetd – DARPA TELNET protocol server

**SYNOPSIS**

**/etc/telnetd**

**DESCRIPTION**

*telnetd* is a server which supports the DARPA standard *TELNET* virtual terminal protocol. *telnetd* is invoked by the internet server (see *inetd(1m)*), normally for requests to connect to the *TELNET* port as indicated by the */etc/services* file (see *services(4)*).

*telnetd* operates by allocating a pseudo-terminal device (see *pty(7)*) 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 remote client and the login process.

When a *TELNET* session is started up, *telnetd* sends *TELNET* options to the client side indicating a willingness to do *remote echo* of characters, to *suppress go ahead*, and to receive *terminal type information* from the remote client. If the remote client is willing, the remote terminal type is propagated in the environment of the created login process. The pseudo-terminal allocated to the client is configured to operate in "cooked" mode, and with XTABS and CRMOD enabled (see *termio(7)*).

*telnetd* is willing to *do*: *echo*, *binary*, *suppress go ahead*, and *timing mark*. *telnetd* is willing to have the remote client *do*: *binary*, *terminal type*, and *suppress go ahead*.

**SEE ALSO**

*telnet(1C)*

**ERRORS**

Some *TELNET* commands are only partially implemented.

The *TELNET* protocol allows for the exchange of the number of lines and columns on the user's terminal, but *telnetd* doesn't make use of them.

Because of bugs in the original 4.2 BSD *telnet(1C)*, *telnetd* performs some dubious protocol exchanges to try to discover if the remote client is, in fact, a 4.2 BSD *telnet(1C)*.

*Binary mode* has no common interpretation except between similar operating systems (UNIX in this case).

The terminal type name received from the remote client is converted to lower case.

The *packet* interface to the pseudo-terminal (see *pty(7)*) should be used for more intelligent flushing of input and output queues.

*telnetd* never sends *TELNET go ahead* commands.

**ORIGIN**

4.3 BSD

**NAME**

tftpd – DARPA Trivial File Transfer Protocol server

**SYNOPSIS**

**/usr/etc/in.tftpd** [ **–d** ] [ *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(4),* and is invoked each time a datagram reaches this port by the internet server *inetd(1M).*

Due to the lack of authentication information, *tftpd* will allow only publicly readable files to be accessed. To do this, **tftpd** executes as *uid –2* , *gid –2* , assuming that no files exist with that owner or group. However, nothing check this assumption or enforces this restriction.

**SEE ALSO**

**ERRORS**

This server is known only to be self consistent (i.e. it operates with the user TFTP program, *tftp(1)*).

**NAME**

tic – terminfo compiler

**SYNOPSIS**

**tic** [**–v**[ *n* ] ] [**–c** ] *file*

**DESCRIPTION**

*tic* translates a *terminfo(4)* file from the source format into the compiled format. The results are placed in the directory */usr/lib/terminfo*. The compiled format is necessary for use with the library routines described in *curses(3X)*.

" .}S 1 3 "**–v**" " "**n**"""" "" "" "" "" ""

(verbose) output to standard error trace information showing *tic*'s progress. The optional integer *n* is a number from 1 to 10, inclusive, indicating the desired level of detail of information. If *n* is omitted, the default level is 1. If *n* is specified and greater than 1, the level of detail is increased.

**–c** only check *file* for errors. Errors in **use=** links are not detected.

*file* contains one or more *terminfo(4)* terminal descriptions in source format (see *terminfo(4)*). Each description in the file describes the capabilities of a particular terminal. When a **use=***entry-name* field is discovered in a terminal entry currently being compiled, *tic* reads in the binary from */usr/lib/terminfo* to complete the entry. (Entries created from *file* will be used first. If the environment variable **TERMINFO** is set, that directory is searched instead of */usr/lib/terminfo*.) **Tic** duplicates the capabilities in *entry-name* for the current entry, with the exception of those capabilities that explicitly are defined in the current entry.

If the environment variable **TERMINFO** is set, the compiled results are placed there instead of */usr/lib/terminfo*.

**FILES**

*/usr/lib/terminfo/?/** compiled terminal description data base

**SEE ALSO**

*curses(3X), term(4), terminfo(4)* in the *Programmer's Reference Manual.*
Chapter 10 in the *Programmer's Guide*.

**WARNINGS**

Total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

Terminal names exceeding 14 characters will be truncated to 14 characters and a warning message will be printed.

When the **–c** option is used, duplicate terminal names will not be diagnosed; however, when **–c** is not used, they will be.

**ERRORS**

To allow existing executables from the previous release of the UNIX System to continue to run with the compiled terminfo entries created by the new terminfo compiler, cancelled capabilities will not be marked as cancelled within the terminfo binary unless the entry name has a '**+**' within it. (Such terminal names are only used for inclusion within other entries via a **use=** entry. Such names would not be used for real terminal names.)

For example:

```
4415+nl, kf1@, kf2@, ....

4415+base, kf1=\EOc, kf2=\EOd, ....

4415-nl|4415 terminal without keys,
    use=4415+nl, use=4415+base,
```

The above example works as expected; the definitions for the keys do not show up in the *4415−nl* entry. However, if the entry *4415+nl* did not have a plus sign within its name, the cancellations would not be marked within the compiled file and the definitions for the function keys would not be cancelled within *4415−nl*.

**DIAGNOSTICS**

Most diagnostic messages produced by *tic* during the compilation of the source file are preceded with the approximate line number and the name of the terminal currently being worked on.

*mkdir* ... returned bad status
: The named directory could not be created.

File does not start with terminal names in column one
: The first thing seen in the file, after comments, must be the list of terminal names.

Token after a *seek(2)* not NAMES
: Somehow the file being compiled changed during the compilation.

Not enough memory for use_list element
: or

Out of memory
: Not enough free memory was available (*malloc(3)* failed).

Can't open ...
: The named file could not be created.

Error in writing ...
: The named file could not be written to.

Can't link ... to ...
: A link failed.

Error in re-reading compiled file ...
: The compiled file could not be read back in.

Premature EOF
: The current entry ended prematurely.

Backspaced off beginning of line
: This error indicates something wrong happened within *tic*.

Unknown Capability - "..."
: The named invalid capability was found within the file.

Wrong type used for capability "..."
: For example, a string capability was given a numeric value.

Unknown token type
: Tokens must be followed by '@' to cancel, ',' for booleans, '#' for numbers, or '=' for strings.

"...": bad term name
: or

Line ...: Illegal terminal name - "..."

The given name was invalid. Names must not contain white space or slashes, and must begin with a letter or digit.

"...": terminal name too long.
An extremely long terminal name was found.

"...": terminal name too short.
A one-letter name was found.

"..." filename too long, truncating to "..."
The given name was truncated to 14 characters due to UNIX file name length limitations.

"..." defined in more than one entry. Entry being used is "...".
An entry was found more than once.

Terminal name "..." synonym for itself
A name was listed twice in the list of synonyms.

At least one synonym should begin with a letter.
At least one of the names of the terminal should begin with a letter.

Illegal character - "..."
The given invalid character was found in the input file.

Newline in middle of terminal name
The trailing comma was probably left off of the list of names.

Missing comma
A comma was missing.

Missing numeric value
The number was missing after a numeric capability.

NULL string value
The proper way to say that a string capability does not exist is to cancel it.

Very long string found. Missing comma?
self-explanatory

Unknown option. Usage is:
An invalid option was entered.

Too many file names. Usage is:
self-explanatory

"..." non-existant or permission denied
The given directory could not be written into.

"..." is not a directory
self-explanatory

"...": Permission denied
access denied.

"...": Not a directory
**tic** wanted to use the given name as a directory, but it already exists as a file.

SYSTEM ERROR!! Fork failed!!!
A *fork(2)* failed.

Error in following up use-links. Either there is a loop in the links or they reference non-existant terminals. The following is a list of the entries involved:

A *terminfo(4)* entry with a **use=***name* capability either referenced a non-existant

terminal called *name* or *name* somehow referred back to the given entry.

**NAME**

tunefs.ffs – tune up an existing file system

**SYNOPSIS**

**/etc/tunefs.ffs** *tuneup-options special | filesys*

**DESCRIPTION**

*tunefs.ffs* is designed to change the dynamic parameters of an FFS file system which affect the layout policies. The parameters which are to be changed are indicated by the flags given below:

**–a** maxcontig

This specifies the maximum number of contiguous blocks that will be laid out before forcing a rotational delay (see –d below). The default value is one, since most device drivers require an interrupt per disk transfer. Device drivers that can chain several buffers together in a single transfer should set this to the maximum chain length.

**–d** rotdelay

This specifies the expected time (in milliseconds) to service a transfer completion interrupt and initiate a new transfer on the same disk. It is used to decide how much rotational spacing to place between successive blocks in a file.

**–e** maxbpg

This indicates the maximum number of blocks any single file can allocate out of a cylinder group before it is forced to begin allocating blocks from another cylinder group. Typically this value is set to about one quarter of the total blocks in a cylinder group. The intent is to prevent any single file from using up all the blocks in a single cylinder group, thus degrading access times for all files subsequently allocated in that cylinder group. The effect of this limit is to cause big files to do long seeks more frequently than if they were allowed to allocate all the blocks in a cylinder group before seeking elsewhere. For file systems with exclusively large files, this parameter should be set higher.

**–m** minfree

This value specifies the percentage of space held back from normal users; the minimum free space threshold. The default value used is 10%. This value can be set to zero, however up to a factor of three in throughput will be lost over the performance obtained at a 10% threshold. Note that if the value is raised above the current usage level, users will be unable to allocate files until enough files have been deleted to get under the higher threshold.

**–o** optimization preference

The file system can either try to minimize the time spent allocating blocks, or it can attempt minimize the space fragmentation on the disk. If the value of minfree (see above) is less than 10%, then the file system should optimize for space to avoid running out of full sized blocks. For values of minfree greater than or equal to 10%, fragmentation is unlikely to be problematical, and the file system can be optimized for time.

**SEE ALSO**

*fs(4FFS), newfs(1FFS), mkfs(1FFS), dumpfs(1FFS)*

M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems 2*, 3. pp 181-197, August 1984 (reprinted in the System Manager's Manual, SMM:14).

**ERRORS**

This program should work on mounted and active file systems. Because the super-block is not kept in the buffer cache, the changes will only take effect if the program is run on dismounted

file systems. To change the root file system, the system must be rebooted after the file system is tuned.

You can tune a file system, but you can't tune a fish.

**NAME**

uadmin – administrative control

**SYNOPSIS**

**/etc/uadmin** *cmd fcn*

**DESCRIPTION**

The *uadmin* command provides control for basic administrative functions. This command is tightly coupled to the System Administration procedures and is not intended for general use. It may be invoked only by the super-user.

The arguments *cmd* (command) and *fcn* (function) are converted to integers and passed to the *uadmin* system call.

**SEE ALSO**

*uadmin(2)* in the *Programmer's Reference Manual.*

**NAME**

unsetenv – unset prom environment variable

**SYNOPSIS**

**unsetenv** var

**DESCRIPTION**

Use the *unsetenv* command to delete an existing environment variable.

**SEE ALSO**

printenv(1prom), setenv(1prom), intro(1spp)

**NAME**

uucheck – check the uucp directories and permissions file

**SYNOPSIS**

**/usr/lib/uucp/uucheck** [ **–v** ] [ **–x** *debug_level* ]

**DESCRIPTION**

*uucheck* checks for the presence of the *uucp* system required files and directories. Within the *uucp* makefile, it is executed before the installation takes place. It also checks for some obvious errors in the Permissions file (*/usr/lib/uucp/Permissions*). When executed with the **–v** option, it gives a detailed explanation of how the uucp programs will interpret the Permissions file. The **–x** option is used for debugging. *Debug-option* is a single digit in the range 1-9; the higher the value, the greater the detail.

Note that *uucheck* can only be used by the super-user or *uucp*.

**FILES**

*/usr/lib/uucp/Systems*
*/usr/lib/uucp/Permissions*
*/usr/lib/uucp/Devices*
*/usr/lib/uucp/Maxuuscheds*
*/usr/lib/uucp/Maxuuxqts*
*/usr/spool/uucp/**
*/usr/spool/locks/LCK**
*/usr/spool/uucppublic/**

**SEE ALSO**

*uucico(1M), uusched(1M).*
*uucp(1C), uustat(1C), uux(1C)* in the *User's Reference Manual.*

**ERRORS**

The program does not check file/directory modes or some errors in the Permissions file such as duplicate login or machine name.

**NAME**

uucico – file transport program for the uucp system

**SYNOPSIS**

**/usr/lib/uucp/uucico** [ **–r** *role_number* ] [ **–x** *debug_level* ]
[ **–i** *interface* ] [ **–d** *spool_directory* ] **–s** *system_name*

**DESCRIPTION**

*uucico* is the file transport program for *uucp* work file transfers. Role numbers for the **–r** are the digit 1 for master mode or 0 for slave mode (default). The **–r** option should be specified as the digit 1 for master mode when **uucico** is started by a program or *cron*. *uux* and *uucp* both queue jobs that will be transferred by **uucico.** It is normally started by the scheduler, *uusched* , but can be started manually; this is done for debugging. For example, the shell *uutry* starts **uucico** with debugging turned on. A single digit must be used for the **–x** option with higher numbers for more debugging. The **–i** option defines the interface used with **uucico**. This interface only affects slave mode. Known interfaces are UNIX (default), TLI (basic Transport Layer Interface), and TLIS (Transport Layer Interface with Streams modules, read/write).

**FILES**

*/usr/lib/uucp/Systems*
*/usr/lib/uucp/Permissions*
*/usr/lib/uucp/Devices*
*/usr/lib/uucp/Devconfig*
*/usr/lib/uucp/Sysfiles*
*/usr/lib/uucp/Maxuuxqts*
*/usr/lib/uucp/Maxuuscheds*
*/usr/spool/uucp/**
*/usr/spool/locks/LCK**
*/usr/spool/uucppublic/**

**SEE ALSO**

*cron(1M), uusched(1M), uutry(1M).*
*uucp(1C), uustat(1C), uux(1C)* in the *User's Reference Manual.*

**NAME**

uucleanup – uucp spool directory clean-up

**SYNOPSIS**

**/usr/lib/uucp/uucleanup** [ **−C***time* ] [ [ **−D***time* ] [ **−W***time* ] [ **−X***time* ] [ **−m***string* ] [ **−o***time* ] [ **−s***system* ]

**DESCRIPTION**

*uucleanup* will scan the spool directories for old files and take appropriate action to remove them in a useful way:

Inform the requestor of send/receive requests for systems that can not be reached.

Return mail, which cannot be delivered, to the sender.

Delete or execute rnews for rnews type files (depending on where the news originated--locally or remotely).

Remove all other files.

In addition, there is provision to warn users of requests that have been waiting for a given number of days (default 1). Note that *uucleanup* will process as if all option *times* were specified to the default values unless *time* is specifically set.

The following options are available.

**−C***time* — Any **C.** files greater or equal to *time* days old will be removed with appropriate information to the requestor. (default 7 days)

**−D***time* — Any **D.** files greater or equal to *time* days old will be removed. An attempt will be made to deliver mail messages and execute rnews when appropriate. (default 7 days)

**−W***time* — Any **C.** files equal to *time* days old will cause a mail message to be sent to the requestor warning about the delay in contacting the remote. The message includes the *JOBID*, and in the case of mail, the mail message. The administrator may include a message line telling whom to call to check the problem (**−m** option). (default 1 day)

**−X***time* — Any **X.** files greater or equal to *time* days old will be removed. The **D.** files are probably not present (if they were, the **X.** could get executed). But if there are **D.** files, they will be taken care of by D. processing. (default 2 days)

**−m***string* — This line will be included in the warning message generated by the **−W** option.

**−o***time* — Other files whose age is more than *time* days will be deleted. (default 2 days) The default line is "See your local administrator to locate the problem".

**−s***system* — Execute for *system* spool directory only.

**−x***debug_level*
The **−x** debug level is a single digit between 0 and 9; higher numbers give more detailed debugging information. (If **uucleanup** was compiled with -DSMALL, no debugging output will be available.)

This program is typically started by the shell *uudemon.cleanup*, which should be started by *cron(1M)*.

**FILES**

| | |
|---|---|
| */usr/lib/uucp* | directory with commands used by *uucleanup* internally |
| */usr/spool/uucp* | spool directory |

**SEE ALSO**

*cron(1M).*
*uucp(1C), uux(1C)* in the *User's Reference Manual.*

**NAME**

uucpd – UUCP network connection daemon

**SYNOPSIS**

**/usr/etc/uucpd**

**DESCRIPTION**

This command is run by the Internet daemon in response to requests made on the "uucp" service port as listed in the file */etc/services*. The resulting connection allows UUCP transfers over the network, removing the requirement for dedicated ports.

This daemon acts as a login server to start UUCP connections. First, it prompts with the word "login:" and waits for the login name to be given. If there is a password for the account (as is advisable), this is prompted for and checked. The account used must have the login shell set to */usr/lib/uucp/uucico* to be considered valid for this type of connection.

Once a valid login has been established, the command */usr/lib/uucp/uucico* is executed to begin the UUCP session.

If it exists, the login accounting file */usr/adm/wtmp* is updated upon establishing the session and upon completion.

**FILES**

| | |
|---|---|
| */etc/services* | Internet services database |
| */etc/inetd.conf* | Internet daemon configuration file |
| */usr/lib/uucp/uucico* | UUCP transfer command |

**SEE ALSO**

*inetd(1m), uucico(1m), uucp(1), services(4).*

**NAME**

uugetty – set terminal type, modes, speed, and line discipline

**SYNOPSIS**

**/usr/lib/uucp/getty** [ **–h** ] [ **–t** *timeout* ] [ **–r** ] *line*
[ *speed* [ *type* [ *linedisc* ] ] ]
**/usr/lib/uucp/getty –c** *file*

**DESCRIPTION**

*uugetty* is identical to *getty(1M)* but changes have been made to support using the line for *uucico*, *cu*, and *ct*; that is, the line can be used in both directions. The **uugetty** will allow users to login, but if the line is free, *uucico*, *cu*, or *ct* can use it for dialing out. The implementation depends on the fact that *uucico*, *cu*, and *ct* create lock files when devices are used. When the "open()" returns (or the first character is read when **–r** option is used), the status of the lock file indicates whether the line is being used by *uucico*, *cu*, *ct*, or someone trying to login. Note that in the **–r** case, several <carriage-return> characters may be required before the login message is output. The human users will be able to handle this slight inconvenience. *Uucico* trying to login will have to be told by using the following login script:

"" \r\d\r\d\r\d\r in:–in: . . .

where the . . . is whatever would normally be used for the login sequence.

An entry for an intelligent modem or direct line that has a **uugetty** on each end must use the **–r** option. (This causes **uugetty** to wait to read a character before it puts out the login message, thus preventing two uugettys from looping.) If there is a *uugetty* on one end of a direct line, there must be a *uugetty* on the other end as well. Here is an */etc/inittab* entry using **uugetty** on an intelligent modem or direct line:

30:2:respawn:/usr/lib/uucp/uugetty –r –t 60 tty12 1200

**FILES**

*/etc/gettydefs*
*/etc/issue*

**SEE ALSO**

*uucico(1M)*, *getty(1M)*, *init(1M)*, *tty(7)*.
*ct(1C)*, *cu(1C)*, *login(1)* in the *User's Reference Manual*.
*ioctl(2)*, *gettydefs(4)*, *inittab(4)* in the *Programmer's Reference Manual*.

**BUGS**

*ct* will not work when **uugetty** is used with an intelligent modem such as penril or ventel.

**NAME**

uusched – the scheduler for the uucp file transport program

**SYNOPSIS**

**/usr/lib/uucp/uusched** [ **–x** *debug_level* ] [ **–u** *debug_level* ]

**DESCRIPTION**

*uusched* is the *uucp* file transport scheduler. It is usually started by the daemon *uudemon.hour* that is started by *cron(1M)* from an entry in */usr/spool/cron/crontab*: 39 * * * * /bin/su uucp -c "/usr/lib/uucp/uudemon.hour > /dev/null" The two options are for debugging purposes only; **–x** *debug_level* will output debugging messages from **uusched** and **–u** *debug_level* will be passed as **–x** *debug_level* to *uucico*. The *debug_level* is a number between 0 and 9; higher numbers give more detailed information.

**FILES**

*/usr/lib/uucp/Systems*
*/usr/lib/uucp/Permissions*
*/usr/lib/uucp/Devices*
*/usr/spool/uucp/**
*/usr/spool/locks/LCK**
*/usr/spool/uucppublic/**

**SEE ALSO**

*cron(1M), uucico(1M).*
*uucp(1C), uustat(1C), uux(1C)* in the *User's Reference Manual.*

**NAME**

uuxqt – execute remote command requests

**SYNOPSIS**

**/usr/lib/uucp/uuxqt** [ **–s** *system* ] [ **–x** *debug_level* ]

**DESCRIPTION**

*uuxqt* is the program that executes remote job requests from remote systems generated by the use of the *uux* command. (*Mail* uses *uux* for remote mail requests). *uuxqt* searches the spool directories looking for *X.* files. For each *X.* file, **uuxqt** checks to see if all the required data files are available and accessible, and file commands are permitted for the requesting system. The *permissions* file is used to validate file accessibility and command execution permission.

There are two environment variables that are set before the *uuxqt* command is executed:
UU_MACHINE is the machine that sent the job (the previous one).
UU_USER is the user that sent the job.
These can be used in writing commands that remote systems can execute to provide information, auditing, or restrictions.

The **-x** *debug_level* is a single digit between 0 and 9. Higher numbers give more detailed debugging information.

**FILES**

*/usr/lib/uucp/Permissions*
*/usr/lib/uucp/Maxuuxqts*
*/usr/spool/uucp/**
*/usr/spool/locks/LCK**

**SEE ALSO**

*uucico(1M).*
*uucp(1C), uustat(1C), uux(1C), mail(1)* in the *User's Reference Manual.*

**NAME**

volcopy – make literal copy of file system

**SYNOPSIS**

**/etc/volcopy** [ *options* ] *fsname srcdevice volname1 destdevice volname2*

**DESCRIPTION**

*volcopy* makes a literal copy of the file system using a blocksize matched to the device. *options* are:

**–a** invoke a verification sequence requiring a positive operator response instead of the standard 10 second delay before the copy is made

**–s** (default) invoke the **DEL if wrong** verification sequence.

The program requests length and density information if it is not given on the command line or is not recorded on an input tape label. If the file system is too large to fit on one reel, **volcopy** will prompt for additional reels. Labels of all reels are checked. Tapes may be mounted alternately on two or more drives. If **volcopy** is interrupted, it will ask if the user wants to quit or wants a shell. In the latter case, the user can perform other operations (e.g., *labelit*) and return to **volcopy** by exiting the new shell.

The *fsname* argument represents the mounted name (e.g.: *root*, *u1*, etc.) of the filsystem being copied.

The *srcdevice* or *destdevice* should be the physical disk section or tape (e.g.: */dev/dsk/cld0s8*, */dev/rdsk/cld1s8*, etc.).

The *volname* is the physical volume name (e.g.: *pk3*, *t0122*, etc.) and should match the external label sticker. Such label names are limited to six or fewer characters. *Volname* may be **–** to use the existing volume name.

*srcdevice* and *volname1* are the device and volume from which the copy of the file system is being extracted. *destdevice* and *volname2* are the target device and volume.

*fsname* and *volname* are recorded in the last 12 characters of the superblock (*char fsname[6], volname[6];*).

**FILES**

*/etc/log/filesave.log* a record of file systems/volumes copied

**SEE ALSO**

*labelit(1M).*
*fs(4)* in the *Programmer's Reference Manual.*
*sh(1)* in the *User's Reference Manual.*

**WARNINGS**

**Volcopy** does not support tape-to-tape copying. Use *dd(1)* for tape-to-tape copying.

NAME

warm – attempt to warm start current image

SYNOPSIS

**warm**

DESCRIPTION

The *warm* command examines memory for a restart block. If a correctly formatted restart block is found, control transfers to the existing memory image at the entry point given in the restart block. A restart block contains information that tells how to re-enter an existing image. Typically, the existing image has earlier aborted or terminated due to a device failure.

SEE ALSO

restart(1spp), prom(1prom)

**NAME**

whodo – who is doing what

**SYNOPSIS**

**/etc/whodo**

**DESCRIPTION**

*whodo* produces formatted and dated output from information in the */etc/utmp* and */etc/ps_data* files. The display is headed by the date, time and machine name. For each user logged in, device name, user-id and login time is shown, followed by a list of active processes associated with the user-id. The list includes the device name, process-id, cpu minutes and seconds used, and process name.

**EXAMPLE**

The command:

```
whodo
```

produces a display like this:

```
Tue Mar 12 15:48:03 1985
bailey

tty09    mcn      8:51
   tty09   28158   0:29 sh

tty52    bdr     15:23
   tty52   21688   0:05 sh
   tty52   22788   0:01 whodo
   tty52   22017   0:03 vi
   tty52   22549   0:01 sh

xt162    lee     10:20
   tty08    6748   0:01 layers
   xt162    6751   0:01 sh
   xt163    6761   0:05 sh
   tty08    6536   0:05 sh
```

**FILES**

*/etc/passwd*
*/etc/ps_data*
*/etc/utmp*

**SEE ALSO**

*ps(1), who(1)* in the *User's 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. It is used by all the 10Mb/s Ethernet interface drivers. It is not specific to Internet protocols or to 10Mb/s Ethernet, but this implementation currently supports only that combination.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

To facilitate communications with systems which do not use ARP , *ioctl*s are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each ioctl takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These ioctls may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
    struct sockaddr         arp_pa;     /* protocol address */
    struct sockaddr         arp_ha;     /* hardware address */
    int                     arp_flags;  /* flags */
};
/*  arp_flags field values */
#define     ATF_COM             0x02    /* completed entry (arp_ha valid) */
#define     ATF_PERM            0x04    /* permanent entry */
#define     ATF_PUBL            0x08    /* publish (respond for other host) */
#define     ATF_USETRAILERS     0x10    /* send trailer packets to host */
```

The address family for the *arp_pa* sockaddr must be AF_INET; for the *arp_ha* sockaddr it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM, ATF_PUBL and AFT_USETRAILERS. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The peculiar nature of the ARP tables may cause the ioctl to fail if more than 8 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines.

This allows a host to act as an "ARP server," which may be useful in convincing an ARP-only machine to talk to a non-ARP1 machine.

ARP is also used to negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts which wish to receive trailer encapsulations so indicate by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The AFT_USETRAILERS flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

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).

**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.

**SEE ALSO**

*enp*(7), *inet*(7F), *arp*(1M), *ifconfig*(1M)

"An Ethernet Address Resolution Protocol," RFC826, Dave Plummer, Network Information Center, SRI.

"Trailer Encapsulations," RFC893, S.J. Leffler and M.J. Karels, Network Information Center, SRI.

**ERRORS**

ARP packets on the Ethernet use only 42 bytes of data; however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

**NAME**

atarpd, ipfree, ddpipmaps – uShare's ATARP daemon and files

**SYNOPSIS**

*atarpd* [ **– d** *debuglevel* ] [ **– p** *port* ] [ **– h** *host* ]

**DESCRIPTION**

*atarpd* is a uShare process that processes ATARP (AppleTalk Address Resolution Protocol) packets. ATARP provides ARP as well as RARP service for address mapping between Ethernet (e.g. 01:02:03:04:05:06), IP (Internet Protocol, e.g. 192.9.200.3) and AT (AppleTalk, e.g. 0.2.3) addresses. *atarpd* also provides the neccessary AppleTalk RTMP, ZIP, and NBP support for machines that have more than one network interface (e.g. bridges) so that the machine may act as a proper AppleTalk bridge. *atarpd* can also provide the extra AppleTalk gateway services on behalf of other machines as well, so that AppleTalk devices will recognize any IP bridge as a proper AppleTalk gateway, even though the bridge is not running uShare (useful on bridges that are not UNIX machines or do not have uShare ported to them).

In general, *atarpd* is started by a uShare startup script: either usstart, usstart.e, or startsrvr. *atarpd* may also be started from the command line (perhaps for debugging?). The command line arguments are:

- **d** *debuglevel*

  Normally *atarpd* will put itself into the background and into its own process group. This flag and its argument, even if *debuglevel* is 0, will toggle an internal flag so that will remain in the foreground and the same process group as the current shell. Notice that the internal flag is TOGGLED, two - **d** debuglevel arguments will set the debuglevel but otherwise behave normally.

- **p** *port*

  Normally *atarpd* binds itself to port 1902. Use the argument to this flag to change the port that *atarpd* binds to. This is an easy way to prevent *atarpd* from processing ATARP packets, but still provide AppleTalk gateway services. Changing this port may cause other processes that use ATARP to fail.

- **h** *host*

  Normally *atarpd* binds itself to IP address INADDR_ANY (see *<netinet/in.h>*). Some administrators may want to make this the loopback address so that *atarpd* will only process ATARP packets that originate from the same machine.

For uShare to work properly, *atarpd* must be running on at least one machine on the internet. *atarpd* may be run on as many machines as an administrator wants. For optimum performance and minimal network traffic, start a normal *atarpd* process on one machine on every network, and start a restricted *atarpd* process (e.g. **-h 127.0.0.1** ) on every other machine that is running uShare.

**atarpd**'s biggest function is to provide IP to AT address mappings. Since IP addresses are four bytes long and AT addresses are only three bytes, some kind of mapping must take place. By default, without any configuration, *atarpd* will map the lower 16 bits of an IP network address to an AT network address and the lower 8 bits (or less if the network's subnet mask has fewer host bits) of an IP host address to an AT node address. For most sites, especially sites that do not already have an installed AppleTalk internet, this default mapping is adequate. However, care must be taken to make sure that this mapping does not create duplicate AppleTalk network numbers, e.g. 192.9.200 and 193.9.200 would both map to the AppleTalk network 9.200. Since the AppleTalk network number can only be 16 bits (two bytes) long and an IP network number can be from seven to 31 bits long, there is no mapping scheme that can avoid creating duplicate AT network numbers in all cases; there are simply many more possible IP network numbers than AT networks. If the default scheme will not work for your site, or you already have an installed AT internet with well known AT network numbers, or you just want to create

your own IP to AT network mappings, you can do so by adding aliases to the file */etc/networks*. */etc/networks* has become a standard UNIX networking file/database to map names to IP network numbers; for more information on this file, please refer to *networks*(4). For *atarpd* to recognize an alias in */etc/networks* as an IP-AT mapping, the alias must begin with AT and immediatly be followed by the AT network number to be used for the corresponding IP network:

```
cnet    192.9.200    AT1
```

The above line in */etc/networks* tells *atarpd* to map IP network 192.9.200 to AT network 1. On some machines, you can configure a network interface to use a subnet mask (see *ifconfig*(1) or *ifconfig*(1M)) and the interface can be queried to provide the subnet mask. If your machine (most System V machines) can do this great. If not, but your net is using subnetting anyway, you can pass the subnet mask to *atarpd* through the same alias:

```
bsubnetc    128.1.2    AT2,MASK255.255.255
```

The above line in */etc/networks* tells *atarpd* to map IP network 128.1.2 to AT network 2. If you aren't using subnetting or you don't know what it is, don't use the MASK feature. The AppleTalk protocol stack only provides for 254 different hosts on a physical network. This is quite a serious limitation if many AppleTalk devices (e.g. Macintoshes) needed to be attached to one physical network. *atarpd* provides a way to allow more than 254 AT hosts on one physical net by mapping more than one AT net to one IP net, provided that the IP net has the address space (e.g. class C IP nets only allow 254 hosts). For example, the class B IP net 128.4 can have 65534 hosts. That's enough address space for 256 AppleTalk nets, so this net can support 65024 AppleTalk hosts that use IPT's ATARP and DDP1IP protocols. These numbers are not recommendations! They are simply the physical upper bounds allowed by the network protocols. Typical pratical limits are much less and very greatly depending upon other physical attributes of the net (e.g. distance between hosts, cable length, cable quality, network traffic, etc...). To map all of these AT nets into the IP net, *atarpd* applys a mask to the IP network number that allows eight bits of host address space. So, for a class B IP net, the mask would be 255.255.255 and IP addresses in the range [128.4.0.0,128.4.0.255] would map by default to AT addresses [4.0.0,4.0.255], IP hosts in [128.4.1.0,128.4.1.255] would map to AT [4.1.0,4.1.255], and so on. An administrator can provide their own mapping (to circumvent **atarpd**'s default mapping) through network aliases in */etc/networks* for this case as well:

```
bnet    128.5    AT7.0-7.255
```

The above line tells *atarpd* to map IP hosts in [128.5.0,128.5.255] to AT [7.0.0,7.0.255], and so on till IP [128.5.255.0,128.5.255.255] is mapped to AT [7.255.0,7.255.255]. The network alias may be a list of AT networks instead of a range or a combination of both:

```
bnet2    128.6    AT3.1,4.5,6,8.3-8.255
```

The above line tells *atarpd* to map the first three "subnets" of IP network 128.6 to AT networks 3.1, 4.5, and 6, while the rest of the IP network is mapped into AT nets 8.3 through 8.255.

If your site is on a class A or B IP net without subnetting or the subnet mask results in more than 256 possible IP hosts (e.g. eight bits of IP host address), then you can use the IP to AT range scheme. But some sites may only have a few uShare hosts and clients on the net and may not want to go through the trouble of setting up the AT net ranges. These sites may add direct IP to AT host mappings in *{/ushare/{etc,etc.local},.}/ddpipmaps* for every host on the

net. The format of this(these) file(s) is:

<ethernet address>, <ip host address>, <at host address>

The Ethernet address is provided for RARP service as well; if RARP is unused or the Ethernet address is unknown, just put a 0 in for <ethernet address>. The other fields must be defined and cannot evaluate to zero! Both <ip host address> and <at host address> may be either numbers or names in */etc/hosts.*

Here are some example entries for the ddpipmaps file(s):

0, 128.9.2.1, 1.1
0, 128.9.3.7, 1.2
0, host1, 1.3
0, host2, host2-at
01:02:03:04:05:06, host3, 1.7

**atarpd**'s second feature is to provide RARP service. For this, *atarpd* looks at entries in the ddpipmaps file(s) described above, the file(s) */ushare/{etc,etc.local}/ipfree,* and */etc/hosts.* The ddpipmaps file(s) is used to provide a hard mapping based on the requesting device's Ethernet address. The ipfree file(s) is used to restrict **atarpd**'s choice of IP address to particular ranges or networks. Finally, */etc/hosts* is consulted to make sure that a potential IP address isn't already defined by another host (if an entry in */etc/hosts* has a name or alias where the first five letters are ATARP, *atarpd* will assume that this address was reserved for it and will still use the address). */etc/hosts* has become a standard UNIX networking file, for documentation on */etc/hosts* please refer to *hosts*(4). An entry in an ipfree file can be a single entry:

128.9.1.2

or a list of IP addresses:

128.9.1.3,128.9.1.5,128.9.1.7

or an IP address range:

128.10.0.1-128.10.255.254

Any line in ipfree may be preceded by 'disable' to tell *atarpd* to ignore RARP requests from a network or networks:

disable 128.9.1.3,128.9.1.5,128.9.1.7

The above line tells *atarpd* to ignore RARP requests that come from IP network 128 (**atarpd** only looks at the network part of an IP address if the 'disable' keyword is present). If an ipfree file does not exist or is empty or *atarpd* receives a RARP request for a network that it serves but is not in an ipfree file, *atarpd* will use the entire address range of the network the request originated on. Reguardless of entries in ipfree, ddpipmaps or */etc/hosts, atarpd* will only choose an IP address if it appears that no other host is using the address by passing ALL of the following checks: The IP address is not already defined to another Ethernet address in **atarpd**'s internal tables. The IP address isn't already mapped to another Ethernet address in the host's ARP tables. Another *atarpd* server does not have a different mapping for the IP address. Another host does not answer an ARP lookup for the IP address. If the IP address is not defined in */etc/hosts.* If the IP address is defined in */etc/hosts,* the name or any alias has the first five letters, ATARP.

The ATARP naming convention in check 6 above is provided so that IP addresses used by **atarpd**'s RARP service can be reserved so that network administrator's unfamiliar with uShare or *atarpd* will not accidently steal IP addresses away from *atarpd* ; even though IP addressee may be reserved in ipfree, *atarpd* will not use an IP address if it is defined in */etc/hosts* but does not pass check 6.

Another, more difficult but preferred, way to restrict *atarpd* RARP processing is to put a single IP address for every IP network that you do not want *atarpd* to service into an ipfree file. Make sure that the IP addresses you chose are already defined in */etc/hosts* and DO NOT have a name or alias where the first five letters are ATARP. This way you can select which networks an *atarpd* process can serve ATARP RARP requests from. Other methods described above simply turn off all ATARP RARP processing.

**FILES**

*/ushare/etc.local/ddpipmaps*
*/ushare/etc/ddpipmaps*
*./ddpipmaps*
*/ushare/etc.local/ipfree*
*/ushare/etc/ipfree*
*./ipfree*
*/etc/hosts*
*/etc/networks*

**SEE ALSO**

*ifconfig*(1M), *arp*(1M), *netstat*(1M), *gethostent*(3N), *getnetent*(3N), *networks*(4), *hosts*(4), *arp*(7).

**DIAGNOSTICS**

When **atarpd** forks, it prints the pid of the child on stdout. Various levels of debugging output is enabled with a non-zero argument to the **-d** flag; the higher the number, the more the output. uShare also provides the program ataq so that ATARP ARP and RARP queries may be generated from the command line and responses printed on stdout. See the uShare documentation on ataq for more information.

**NAME**

clone – open any minor device on a STREAMS driver

**DESCRIPTION**

*clone* is a STREAMS software driver that finds and opens an unused minor device on another STREAMS driver. The minor device passed to *clone* during the open is interpreted as the major device number of another STREAMS driver for which an unused minor device is to be obtained. Each such open results in a separate *stream* to a previously unused minor device. The *clone* driver consists solely of an open function. This open function performs all of the necessary work so that subsequent system calls (including *close(2)*) require no further involvement of *clone*. *clone* will generate an ENXIO error, without opening the device, if the minor device number provided does not correspond to a valid major device, or if the driver indicated is not a STREAMS driver.

**CAVEATS**

Multiple opens of the same minor device cannot be done through the *clone* interface. Executing *stat(2)* on the file system node for a cloned device yields a different result from executing *fstat(2)* using a file descriptor obtained from opening the node.

**SEE ALSO**

*log(7)*.
*STREAMS Programmer's Guide*.

**NAME**

console – console interface

**DESCRIPTION**

The console provides the operator interface to the computer.

The file */dev/console* is the system console, and refers to an asynchronous serial data line originating from the system board. This special file implements the features described in *termio(7)*.

The file */dev/contty* refers to a second asynchronous serial data line originating from the system board. This special file implements the features described in *termio(7)*.

**FILES**

*/dev/console*
*/dev/contty*

**SEE ALSO**

*termio(7)*.

**NAME**

console – Advant*edge* system console interface

**DESCRIPTION**

Usually the console device is a normal tty that also receives system error messages. On Advant*edge* systems, the console is a pseudo tty. If a video board is available a special version of xterm(1) will be started that opens the master side of this tty accepting all messages and displaying them in a window. Without a video board the console will be attached to a normal serial device. Only xterm(1) needs to know that /dev/console is special, all programs may treat it like a normal tty(7).

**FILES**

/dev/console

**NAME**

cp – Integrated Solutions Communications Processor

**SYNOPSIS**

**device cp0 at vme? csr 0xfff520 am 0x3d vector cpintr**

**DESCRIPTION**

An ISI Communications Processor provides 8 or 16 communication lines; and it can also provide printer support for either centronix or data products type printer devices.

Each line attached to the ISI communications processor behaves as described in *tty(7)*. Input and output for each line may independently be set to run at any of a number of speeds; see *tty(7)* for the encoding.

**FILES**

*/dev/tty[h-i][0-9a-f]*

**SEE ALSO**

*tty(7)*

VME-ICP16/8 Intelligent Communications Processor Hardware Reference Manual

**DIAGNOSTICS**

"cp%d: silo overflow." The character silo overflowed before it could be serviced.

"cp%d: line %d overflow."

**NOTES**

The driver currently does not make full use of the hardware capabilities of the ISI Communications Processor, for dealing with printers for example.

**NAME**

dkip – Interphase V-SMD 3200 disk controller interface

**SYNOPSIS**

**controller dkipc0 at vme? csr 0x8600 am 0x2d vector dkipintr**
**disk dkip0 at dkipc0 drive 0**
**disk dkip1 at dkipc0 drive 1**

**DESCRIPTION**

This is a driver for the Interphase V-SMD 3200 disk controller and for other compatible controllers. Files with minor device numbers 0 through 15 refer to various portions of drive 0; minor devices 16 through 31 refer to drive 1, etc. The standard device names begin with "ip" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-4.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface that provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O, buffers should be page aligned for best performance and I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

Disk volumes on MIPS computers systems contain a volume header that describes the contents of the disk and parameters of the physical disk drive. The volume header is a block located at the beginning of all disk media. It contains information about physical device parameters and logical partition information. Refer to *dvh(4)* for details on the disk volume header format. Volume headers are created by formatters and, may be manipulated by *dvhtool(1M)* via the special raw device "rip?vh". Another special raw device exists, "rip?vol", for use by formatters to access the entire disk volume.

**DISK SUPPORT**

This driver configures the drive type of each drive, during initialization, based on the information in the volume header for the particular device. The partition table is also contained in the volume header.

The ip?a partition is normally used for the root file system, and the ip?b partition as a paging area. The ip?c partition maps all the UNIX file system partitions, while rip?vol maps the entire disk.

**FILES**

*/dev/dsk/ips[0-1]d[0-1]s[0-f]*
*/dev/dsk/ips[0-1]d[0-1]vh*
*/dev/rdsk/ips[0-1]d[0-1]vh*

**SEE ALSO**

*dvh(7), dvhtool(1M), prtutoc(1m)*
V/SMD 3200 Disk Controller/Formatter User's Guide

**DIAGNOSTICS**

*ips%d: firmware prom id 0x%x%x (%x/%x/%x)[, no scatter-gather][, no cache].* During initialization, the firmware revision id and date of release are indicated. Also, this informs if scatter-gather or disk cacheing is supported.

*ips%dd%d: no volume header found.* The disk media does not contain a MIPS volume header and has probably not been formatted. No reads or writes may be done on the disk until it has been formatted.

*ips%dd%d: spurious interrupt, csr= %r.* An interrupt was detected from the controller but the status register does not show a valid interrupt condition.

*ips%dd%d: timeout, bnc= %d, cmd=0x%x, usr=0x%x.* An interrupt was not received from the controller in the amount of time expected and the status register is dumped. This indicates a hardware or software failure.

*ips%d: power up diag never completed.* The controller was not able to successfully do its power up diagnostic sequence. This indicates a hardware failure.

*ips%dd%d: error csr= 0x%x, bn=%d, statcode=%x.* An error was encountered during execution on a command. The reason for the error is indicated as well as the current status of each drive.

*ips%dd%d: error table overflowed for lbn%d.* The number of sectors for which errors have been detected has overflowed its table. This indicates serious problems as the error rate is far above what is normal.

*ips%dd%d): error table full, failed to clear drive fault.* The driver attempted to clear a drive fault and was unsuccessful. This indicates a hardware failure. This indicates serious problems as the error rate is far above what is normal.

*ips%d: controller not available.* Could not find the controller where it was specified.

*ips%d: unknown controller type.* Controller found is not a controller that the software supports.

*ips%d:reset failed.* Reset of the controller failed.

*ips%d: can't get controller information.* Could not read controller firmware revision.

*ips%dd%d: volume header contains bad parameters.* The drive did not configure using information in the volume header.

*ips%dd%d: %d/%d/%d, %d:1 interleave.* Gives unit number, cylinders, tracks, sectors, and interleave.

*ips%dd%d: spurious status change interrupt.* Says a spurious status change interrupt was received.

*ips%dd%d: scintr unknown drive status.* Got an unknown failure condition.

*dkipustart: ctlr never acknowledged.* Never acknowledged the receipt of command.

*ips%dd%d: Not enough memory for %s.* Kernel malloc failed to get memory.

*ips%dd%d: I/O error read %s.* Buffer used was marked with error. Last Command probably failed.

*ips%d: old firmware.* Firmware in controller NOT supported by driver.

*ips%d: old 3200 firmware.* Firmware in controller NOT supported by driver.

*ips%dd%d: seek timeout. bn=%d, cmd=0x%x, csr=0x%x.* Indicated that a seek took too long to complete.

*drive fault: A drive fault has occurred.*

**BUGS**

The logged error information should be saved somewhere in the volume directory.

A program to analyze the logged error information (even in its present reduced form) is needed.

Overlapped seeks should be supported.

**NAME**

dksd – general SCSI disk interface.

**SYNOPSIS**

**VECTOR: module=sha vector=0xD ipl=2 unit=6 base=060**

**DESCRIPTION**

This is a general SCSI disk driver and should work with most drives that have embedded controllers. Files with minor device numbers 0 through 15 refer to various portions of target0, lun0; minor devices 16 through 31 refer to target0, lun1, etc. The driver uses the standard System V naming convention.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface that provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O, buffers should be page aligned for best performance and I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

Disk volumes on ISI computers systems contain a volume header that describes the contents of the disk and parameters of the physical disk drive. The volume header is a block located at the beginning of all disk media. It contains information about physical device parameters and logical partitions. Refer to *dvh(5)* for details on the disk volume header format. Volume headers are created by formatters and, may be manipulated by *dvhtool*(8) via the special raw device "rsd?vh". Another special raw device exists, "rsd?vol" for use by formatters to access the entire disk volume.

**DISK SUPPORT**

This driver configures the drive type of each drive, during initialization, based on the information in the volume header for the particular device. The partition table is also contained in the volume header.

The "sds?d0s0" partition is normally used for the root file system, and the "sd?d0s1" partition as a paging area. The "sds?d0s2" partition maps all the UNIX file system partitions, while "sd?vol" maps the entire disk.

**FILES**

/dev/sds[0-4]d0s[0-7] block files
/dev/rsds[0-4]d0s[0-7] raw files
/dev/sd[0-4]vh volume header partition
/dev/sd[0-4]vol entire volume partition

**SEE ALSO**

dvh(5), dvhtool(8), format(8)

**DIAGNOSTICS**

**sense failed: %s: scsi %b** An error occurred during a sense command. The sense commands are normally issued only when some other command like a read or write failed.

**no more retries** When an error has occurred the driver will retry the command up to 3 times.

**ERRORS**

The logged error information should be saved somewhere in the volume directory.

A program to analyze the logged error information (even in its present reduced form) is needed.

**NAME**

enp – CMC 10 Mb/s Ethernet interface

**SYNOPSIS**

**device enp0 at vme? csr 0xde0000 am 0x3d vector enpintr**
**device enp1 at vme? csr 0xe00000 am 0x3d vector enpintr**

**DESCRIPTION**

The *enp* interface provides access to a 10 Mb/s Ethernet network through a CMC controller.

Each of the host's network addresses is specified at boot time with an SIOCSIFADDR ioctl. The station address is discovered by probing the on-board Ethernet address register, and verifies the protocol address. No packets will be sent or accepted until a network address is supplied. The *enp* interface employs the address resolution protocol described in *arp(4P)* to dynamically map between Internet and Ethernet addresses on the local network.

The interface handles both Internet and NS protocol families. The use of trailers is negotiated with ARP. However, this negotiation may be disabled, on a per-interface basis, by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl.

**DIAGNOSTICS**

*enp%d: hardware address %s.* This is a normal autoconfiguration message noting the 6 byte physical ethernet address of the adapter.

*enp%d: can't handle af%d.* The interface was handed a message with addresses formatted in an unsuitable address family; the packet was dropped.

The following messages indicate a probable hardware error performing the indicated operation during autoconfiguration or initialization. See the hardware manual for details.

*enp%d: detected error on reset.*

*enp%d: timed out waiting for reset.*

*enp%d: application firmware failed.*

**SEE ALSO**

*intro(4N), inet(4F), arp(4P)*
ENP-10 User's Manual

**NAME**

fl – floppy disk drive controller interface

**SYNOPSIS**

**VECTOR: module=sha vector=0xF ipl=2 unit=8 base=100**
**VECTOR: module=sha vector=0x10 ipl=2 unit=9 base=110**

**DESCRIPTION**

The floppy disk drive controller is a Western Digital 37C65 chip, implemented as a Pseudo-SCSI device. It responds to a subset of SCSI CCS commands, existing as SCSI devices 8 and 9. A mode select command configures the chip to control the different types of floppy drives available. ISI computers are configured with a 3-1/2 inch high-capacity floppy disk.

In raw I/O, buffers should be page aligned for best performance and I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

Disk volumes on ISI computers systems contain a volume header that describes the contents of the disk and parameters of the physical disk drive. The volume header is a block located at the beginning of all disk media. It contains information about physical device parameters and logical partitions. Refer to *dvh(5)* for details on the disk volume header format. Volume headers are created by formatters and, may be manipulated by *dvhtool*(8) via the special raw device "rfd?t?vh". Another special raw device exists, "rfd?t?vol" for use by formatters to access the entire disk volume.

**DISK SUPPORT**

This driver configures the drive type of each drive during open, based on the minor number of the particular device opened. Once a drive has been opened with a particular drive type, it cannot be opened with a different type until the last process has closed it.

Three drive types are currently configured into the driver, as follows:

Type 0: Low Capacity Media in a High Capacity 3-1/2" drive
Type 1: High Capacity Media in a High Capacity 3-1/2" drive
Type 2: Low Capacity Media in a Low Capacity 5-1/4" drive

Unconfigured types (3-7) can be set with the FLIOCMODSLCT ioctl call. Current configuration details may be obtained with the FLIOCMODSNS ioctl call.

The partition table is contained in the volume header. The "fd?t?a" partition is normally used as the only UNIX file system to maximize space (minimize wasted overhead). The "rfd?t?c" partition maps all the UNIX file system partitions, while "rfd?t?vol" maps the entire disk.

**FILES**

/dev/fd/fd[01]t[0-7][a-h] block files
/dev/fd/rfd[01]t[0-7][a-h] raw files
/dev/fd/rfd[01]t[0-7]vh volume header partition
/dev/fd/rfd[01]t[0-7]vol entire volume partition

**SEE ALSO**

dvh(5), dvhtool(8), flfo(8)

**DIAGNOSTICS**

fl%d: sense failed: %s: scsi %b

fl%d: invalid sense class %d
Controller could not get information about error from drive.

fl%d: %s: code %x: bn %d (0x%x): softerr

fl%d: %s: code %x: bn %d (0x%x): reissued

fl%d: %s: code %x: bn %d (0x%x):

Mostly self-explanatory.
code %x is one of:

| | |
|---|---|
| 0x10 | id CRC error |
| 0x12 | missing id address mark |
| 0x13 | missing data address mark |
| 0x15 | seek positioning error |
| 0x20 | invalid command code |
| 0x21 | invalid logical block addr |
| 0x24 | invalid command descriptor block |
| 0x25 | invalid logical unit number |
| 0x26 | invalid field in parameter list |
| 0x27 | write protect error |
| 0x31 | media format corrupt |
| 0x44 | internal hardware error |

bn %d (0x%x) is the logical block number on the disk where the error occurred, if known.
When an error has occurred the driver will retry the command up to 3 times.

**ERRORS**

The logged error information should be saved somewhere in the volume directory.

A program to analyze the logged error information (even in its present reduced form) is needed.

**NAME**

flformat – raw floppy disk device

**SYNOPSIS**

flformat -[m][f][v][i ilv]

**DESCRIPTION**

raw_floppy_disk_device must be one of the -vol devices.

**OPTIONS**

-m flformat will prompt for mode characteristics, i.e.:

"Number of cylinders per disk"
"Number of bytes per sector"
"Number of sectors per track"
"Number of drive heads (surfaces)"
"Transfer rate"
"Normal gap length"
"Format gap length"
"Motor on delay"
"Motor off delay"
"Head settle delay in ms"
"head step rate in ms"
"Head load time in 2ms units"
"Head unload time in 16ms units"
"MFM encoding (non-zero)"
"High capacity drive (non-zero)"

and set that mode before doing format or writing volume header. raw_floppy_disk_device must be the changeable mode device.

-f execute the format phase

-i ilv when executing format phase, use an interleave factor of "ilv"

-v place default volume header on first cylinder of formatted diskq

**NAME**

icmp – Internet Control Message Protocol

**SYNOPSIS**

None; included automatically with *inet*(7F).

**DESCRIPTION**

The Internet Control Message Protocol, ICMP, is used by gateways and destination hosts which process datagrams to communicate errors in datagram-processing to source hosts. ICMP uses the basic support of IP as if it were a higher level protocol; however, ICMP is actually an integral part of IP. ICMP messages are sent in several situations; for example: when a datagram cannot reach its destination, when the gateway does not have the buffering capacity to forward a datagram, and when the gateway can direct the host to send traffic on a shorter route.

The Internet protocol is not designed to be absolutely reliable. The purpose of these control messages is to provide feedback about problems in the communication environment, not to make IP reliable. There are still no guarantees that a datagram will be delivered or that a control message will be returned. Some datagrams may still be undelivered without any report of their loss. The higher level protocols which use IP must implement their own reliability mechanisms if reliable communication is required.

The ICMP messages typically report errors in the processing of datagrams; for fragmented datagrams, ICMP messages are sent only about errors in handling fragment 0 of the datagram. To avoid the infinite regress of messages about messages etc., no ICMP messages are sent about ICMP messages. ICMP may however be sent in response to ICMP messages (for example, ECHOREPLY). There are eleven types of ICMP packets which can be received by the system. They are defined in this excerpt from <netinet/ip_icmp.h>, which also defines the values of some additional codes specifying the cause of certain errors.

```
/*
 * Definition of type and code field values
 */
#define ICMP_ECHOREPLY              0   /* echo reply */
#define ICMP_UNREACH                3   /* dest unreachable, codes: */
#define ICMP_UNREACH_NET            0   /* bad net */
#define ICMP_UNREACH_HOST           1   /* bad host */
#define ICMP_UNREACH_PROTOCOL       2   /* bad protocol */
#define ICMP_UNREACH_PORT           3   /* bad port */
#define ICMP_UNREACH_NEEDFRAG       4   /* IP_DF caused drop */
#define ICMP_UNREACH_SRCFAIL        5   /* src route failed */
#define ICMP_SOURCEQUENCH/s+1       4   /* packet lost, slow down */
#define ICMP_REDIRECT               5   /* shorter route, codes: */
#define ICMP_REDIRECT_NET           0   /* for network */
#define ICMP_REDIRECT_HOST          1   /* for host */
#define ICMP_REDIRECT_TOSNET        2   /* for tos and net */
#define ICMP_REDIRECT_TOSHOST       3   /* for tos and host */
#define ICMP_ECHO                   8   /* echo service */
#define ICMP_TIMXCEED               11  /* time exceeded, code: */
#define ICMP_TIMXCEED_INTRANS       0   /* ttl==0 in transit */
#define ICMP_TIMXCEED_REASS         1   /* ttl==0 in reass */
#define ICMP_PARAMPROB              12  /* ip header bad */
#define ICMP_TSTAMP                 13  /* timestamp request */
#define ICMP_TSTAMPREPLY            14  /* timestamp reply */
#define ICMP_IREQ                   15  /* information request */
```

```
#define ICMP_IREQREPLY          16  /* information reply */
```

Arriving ECHO and TSTAMP packets cause the system to generate ECHOREPLY and TSTAMPREPLY packets. IREQ packets are not yet processed by the system, and are discarded. UNREACH, SOURCEQUENCH, TIMXCEED and PARAMPROB packets are processed internally by the protocols implemented in the system, or reflected to the user if a raw socket is being used. REDIRECT, ECHOREPLY, TSTAMPREPLY and IREQREPLY are also reflected to users of raw sockets. In addition, REDIRECT messages cause the kernel routing tables to be updated.

**SEE ALSO**

*inet*(7F)

Internet Control Message Protocol, RFC792, J. Postel, USC-ISI (Sun 800-1064-01)

**ERRORS**

IREQ messages are not processed properly: the address fields are not set.

Messages which are source routed are not sent back using inverted source routes, but rather go back through the normal routing mechanisms.

**NAME**

idp – Xerox Internet Datagram Protocol

**SYNOPSIS**

```
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>

s = socket(AF_NS, SOCK_DGRAM, 0);
```

**DESCRIPTION**

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

Xerox protocols are built vertically on top of IDP. Thus, IDP address formats are identical to those used by SPP. Note that the IDP port space is the same as the SPP port space (i.e. a IDP port may be "connected" to a SPP port, with certain options enabled below). 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.

**DIAGNOSTICS**

A socket operation may fail with one of the following errors returned:

[EISCONN] when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;

[ENOTCONN] when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;

[ENOBUFS] when the system runs out of memory for an internal data structure;

[EADDRINUSE] when an attempt is made to create a socket with a port which has already been allocated;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

**SOCKET OPTIONS**

[SO_HEADERS_ON_INPUT]

When set, the first 30 bytes of any data returned from a read or recv from will be the initial 30 bytes of the IDP packet, as described by

```
struct idp {
        u_short         idp_sum;
        u_short         idp_len;
        u_char          idp_tc;
        u_char          idp_pt;
        struct ns_addr  idp_dna;
        struct ns_addr  idp_sna;
};
```

This allows the user to determine the packet type, and whether the packet was a multi-cast packet or directed specifically at the local host. When requested, gives the current state of the option, (NSP_RAWIN or 0).

[SO_HEADERS_ON_OUTPUT]

When set, the first 30 bytes of any data sent will be the initial 30 bytes of

the IDP packet. This allows the user to determine the packet type, and whether the packet should be multi-cast packet or directed specifically at the local host. You can also misrepresent the sender of the packet. When requested, gives the current state of the option. (NSP_RAWOUT or 0).

[SO_DEFAULT_HEADERS]
The user provides the kernel an IDP header, from which it gleans the Packet Type. When requested, the kernel will provide an IDP header, showing the default packet type, and local and foreign addresses, if connected.

[SO_ALL_PACKETS] When set, this option defeats automatic processing of Error packets, and Sequence Protocol packets.

[SO_SEQNO] When requested, this returns a sequence number which is not likely to be repeated until the machine crashes or a very long time has passed. It is useful in constructing Packet Exchange Protocol packets.

**SEE ALSO**

*send*(2), *recv*(2), *intro*(7N).

**NAME**

if – general properties of network interfaces

**DESCRIPTION**

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*(7), do not.

At boot time each interface which has underlying hardware support makes itself known to the system during the autoconfiguration process. 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 (for example, 10Mb/s Ethernets using *arp*(7P)), 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 *ifreq* structure as its parameter. This structure has the form

```
struct  ifreq {
  char  ifr_name[16];      /* name of interface (e.g. "ec0") */
  union {
    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**

*arp*(7P), *lo*(7)

**NAME**

imp – 1822 network interface

**SYNOPSIS**

**pseudo-device imp [ count ]**

**DESCRIPTION**

The *imp* interface, as described in BBN Report 1822, provides access to an intelligent message processor normally used when participating in the Department of Defense ARPA network. The network interface communicates through a device controller, usually an ACC LH/DH or HDH or a DEC IMP-11A, with the IMP. The interface is "reliable" and "flow-controlled" by the host-IMP protocol.

To configure IMP support, at least one of *acc*, *css* or *hdh* must be included. The optional *count* specifies the total number of IMP connections. The network number on which the interface resides is specified at boot time using the SIOCSIFADDR ioctl. The host number is discovered through receipt of NOOP messages from the IMP.

The network interface is always in one of four states: up, down, initializing, or going down. When the system is booted, the interface is marked down. If the hardware controller is successfully probed, the interface enters the initializing state and transmits three NOOP messages to the IMP. It then waits for the IMP to respond with two or more NOOP messages in reply. When it receives these messages it enters the up state. The "going down" state is entered only when notified by the IMP of an impending shutdown. Packets may be sent through the interface only while it is in the up state. Outgoing packets are dropped with the error ENETDOWN returned to the caller if the interface is in any other state.

**DIAGNOSTICS**

**imp%d: not configured.** A hardware interface could not be attached during autoconfiguration because too few IMP pseudo-devices were configured.

**imp%d: leader error.** The IMP reported an error in a leader (1822 message header). This causes the interface to be reset and any packets queued up for transmission to be purged.

**imp%d: going down in 30 seconds.**
**imp%d: going down for hardware PM.**
**imp%d: going down for reload software.**
**imp%d: going down for emergency reset.** The Network Control Center (NCC) is manipulating the IMP. By convention these messages are reported to all hosts on an IMP.

**imp?: host %x, lost %d rfnms.** The IMP had messages outstanding to the host listed, but no RFNM (Request for Next Message) messages were received from the IMP in 127 seconds. The software state for that host is reinitialized.

**imp%d: interface reset.** The host has received an interface reset message from the IMP.

**imp%d: address reset to x%x (%d/%d).** The host has received a NOOP message which caused it to reset its notion of its current address. The Internet address is printed in hexadecimal, with the host and IMP numbers following. This indicates that the address originally set by *ifconfig*(1M) was incorrect, that the IMP has undergone an identity crisis, or that communication between the IMP and the host is being garbled.

**imp%d: data error.** The IMP noted an error in data transmitted. The host-IMP interface is reset and the host enters the init state (awaiting NOOP messages).

**imp%d: interface reset.** The reset process has been completed.

**imp%d: marked down.** After receiving a "going down in 30 seconds" message, and waiting 30 seconds, the host has marked the IMP unavailable. Before packets may be sent to the IMP again, the IMP must notify the host, through a series of NOOP messages, that it is back up.

**imp%d: can't handle af%d.** The interface was handed a message with addresses formatting in an unsuitable address family; the packet was dropped.

**SEE ALSO**

*intro*(7N), *inet*(7F)

**NAME**

imp – IMP raw socket interface

**SYNOPSIS**

**#include <sys/socket.h>**
**#include <netinet/in.h>**
**#include <netimp/if_imp.h>**

**s = socket(AF_IMPLINK, SOCK_RAW, proto);**

**DESCRIPTION**

The raw imp socket provides direct access to the *imp*(7) network interface. Users send packets through the interface using the *send*(2) calls, and receive packets with the *recv*(2), calls. All outgoing packets must have an 1822 96-bit leader on the front. Likewise, packets received by the user will have this leader on the front. The 1822 leader and the legal values for the various fields are defined in the include file *<netimp/if_imp.h>*. The raw imp interface automatically installs the length and destination address in the 1822 leader of all outgoing packets; these need not be filled in by the user.

If the protocol selected, *proto*, is zero, the socket will receive all IMP messages except RFNM and incompletes which are not input data for a kernel protocol. If *proto* is non-zero, only messages for the specified link type will be received.

**DIAGNOSTICS**

An operation on a socket may fail with one of the following errors:

| | |
|---|---|
| [EISCONN] | when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected; |
| [ENOTCONN] | when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected; |
| [ENOBUFS] | when the system runs out of memory for an internal data structure; |
| [ENOBUFS] | eight messages to the destination host are outstanding, and another eight are already queued for output; |
| [EADDRNOTAVAIL] | when an attempt is made to create a socket with a network address for which no network interface exists. |

**SEE ALSO**

*intro*(7N), *inet*(7F), *imp*(7)

**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) transport layer, and utilizing the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP protocol.

**ADDRESSING**

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

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

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

Sockets may be created with the local address INADDR_ANY to effect "wildcard" matching on incoming messages. The address in a *connect(2)* or *sendto(2)* call may be given as INADDR_ANY to mean "this host." The distinguished address INADDR_BROADCASTs+1 is allowed as a shorthand for the broadcast address on the primary network if the first network configured supports broadcast.

**PROTOCOLS**

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

The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the most-significant bit is clear in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses use the high-order 16 bits as the network field, and Class C addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to the DARPA Internet may chose to use a single network number for the cluster; this is done by using subnet addressing. The local (host) portion of the address is further subdivided into subnet and host parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry. Subnet addressing is enabled and examined by the following *ioctl(2)* commands on a datagram socket in the Internet domain; they have the same form as the SIOCIFADDR command (see *intro(7N)*).

SIOCSIFNETMASK  Set interface network mask. The network mask defines the network part of the address; if it contains more of the address than the address type would indicate, then subnets are in use.

SIOCGIFNETMASK  Get interface network mask.

**SEE ALSO**

*ioctl*(2), *socket*(2) in the *Programmer's Reference Manual.*
*icmp*(7P), *intro*(7N), *tcp*(7), *udp*(7P).

The chapters on Interprocess Communication (15, 16, and 17) in the *Programmer's Reference Manual.*

**CAVEAT**

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

**NAME**

intro – introduction to special files

**DESCRIPTION**

This section describes various special files that refer to specific hardware peripherals, and UNIX system device drivers. STREAMS [see *intro*(2)] software drivers, modules and the STREAMS-generic set of *ioctl*(2) system calls are also described. For hardware related files, the names of the entries are generally derived from names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding UNIX system device driver are discussed where applicable.

Disk device file names are in the following format:

**/dev/{r}dsk/c#d#s#**

where **r** indicates a raw interface to the disk, the **c#** indicates the controller number, **d#** indicates the device attached to the controller and **s#** indicates the section number of the partitioned device.

**SEE ALSO**

*Disk/Tape Management* in the *System Administrator's Guide*.

**NAME**

networking – introduction to networking facilities

**SYNOPSIS**

**#include <sys/socket.h>**
**#include <net/route.h>**
**#include <net/if.h>**

**DESCRIPTION**

This section briefly describes the networking facilities available in the system. Documentation in this part of section 7 is broken up into three areas: *protocol families* (domains), *protocols*, and *network interfaces*. Entries describing a protocol family are marked "7F," while entries describing protocol use are marked "7P." Hardware support for network interfaces are found among the standard "7" entries.

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, one per *socket(2)* type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in *socket(2)*. A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol specific. All protocols 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.

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. The SYNOPSIS section of each network interface entry gives a sample specification of the related drivers for use in providing a system description to *inetd*(1M). The DIAGNOSTICS section lists messages which may appear on the console and/or in the system error log, */usr/adm/messages,* due to errors in device operation.

**PROTOCOLS**

The system currently supports the DARPA Internet protocols and the Xerox Network Systems(tm) protocols. Raw socket interfaces are provided to the IP protocol layer of the DARPA Internet, to the IMP link layer (1822), and to the IDP protocol of Xerox NS. Consult the appropriate manual pages in this section for more information regarding the support for each protocol family.

**ADDRESSING**

Associated with each protocol family is an address format. The following address formats are used by the system (and additional formats are defined for possible future implementation):

```
#define  AF_UNIX        1     /* local to host (pipes, portals) */
#define  AF_INET        2     /* internetwork: UDP, TCP, etc. */
#define  AF_IMPLINK     3     /* arpanet imp addresses */
#define  AF_PUP         4     /* pup protocols: e.g. BSP */
```

```
#define  AF_NS          6     /* Xerox NS protocols */
#define  AF_HYLINK      15    /* NSC Hyperchannel */
```

**ROUTING**

The network facilities provided 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 socket-specific *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 super-user.

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) */
#define  RTF_DYNAMIC     0x10  /* created dynamically (by redirect) */
```

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 and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. Normally the protocol specifies the route through each interface as a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (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 routing entry will be marked down and removed from the routing table, but the resources associated with it will not be reclaimed until all references to it are released. The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent 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.

When routing a packet, the kernel will first attempt to find a route to the destination host. Failing that, a search is made for a route to the network of the destination. Finally, any route to a default ("wildcard") gateway is chosen. If multiple routes are present in the table, the first route found will be used. If no entry is found, the destination is declared to be unreachable.

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(7)*, do not.

The following *ioctl* calls may be used to manipulate network interfaces. The *ioctl* is made on a socket (typically of type SOCK_DGRAM) in the desired domain. 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 (e.g. "ec0") */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                int     ifru_metric;
        } 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_broadaddr   ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags       ifr_ifru.ifru_flags     /* flags */
#define ifr_metric      ifr_ifru.ifru_metric    /* routing metric */
};
```

SIOCSIFADDR — Set interface address for protocol family. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR — Get interface address for protocol family.

SIOCSIFDSTADDR — Set point to point address for protocol family and interface.

SIOCGIFDSTADDR — Get point to point address for protocol family and interface.

SIOCSIFBRDADDR — Set broadcast address for protocol family and interface.

SIOCGIFBRDADDR — Get broadcast address for protocol family and interface.

SIOCSIFFLAGS — Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified; some interfaces may be reset so that incoming packets are no longer received. When marked up again, the interface is reinitialized.

SIOCGIFFLAGS — Get interface flags.

SIOCSIFMETRIC — Set interface routing metric. The metric is used only by user-level routers.

SIOCGIFMETRIC — Get interface metric.

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 SIOCGIFCON F 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(2), ioctl(2), intro(7), routed(1M)*

**NAME**

ip – Internet Protocol

**SYNOPSIS**

None; included by default with *inet*(4F).

**DESCRIPTION**

The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. It provides for transmitting blocks of data called "datagrams" from sources to destinations, where sources and destinations are hosts identified by fixed-length addresses. It also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks.

IP is specifically limited in scope. There are no mechanisms to augment end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols. IP can capitalize on the services of its supporting networks to provide various types and qualities of service.

IP is called on by host-to-host protocols, including *tcp*(4P) a reliable stream protocol, *udp*(4P) a socket-socket datagram protocol, and *nd*(4P) the network disk protocol. Other protocols may be layered on top of IP using the *raw* protocol facilities described here to receive and send datagrams with a specific IP protocol number. The IP protocol calls on local network drivers to carry the internet datagram to the next gateway or destination host.

When a datagram arrives at a UNIX system host, the system performs a checksum on the header of the datagram. If this fails, or if the datagram is unreasonably short or the header length specified in the datagram is not within range, then the datagram is dropped. Checksumming of Internet datagrams may be disabled for debugging purposes by patching the kernel variable *ipcksum* to have the value 0.

Next the system scans the IP options of the datagram. Options allowing for source routing (see *routing*(4N)) and also the collection of time stamps as a packet follows a particular route (for network monitoring and statistics gathering purposes) are handled; other options are ignored. Processing of source routing options may result in an UNREACH *icmp*(4P) message because the source routed host is not accessible.

After processing the options, IP checks to see if the current machine is the destination for the datagram. If not, then IP attempts to forward the datagram to the proper host. Before forwarding the datagram, IP decrements the time to live field of the datagram by IPTTLDEC seconds (currently 5 from <netinet/ip.h>), and discards the datagram if its lifetime has expired, sending an ICMP TIMXCEED error packet back to the source host. Similarly if the attempt to forward the datagram fails, then ICMP messages indicating an unreachable network, datagram too large, unreachable port (datagram would have required broadcasting on the target interface, and IP does not allow directed broadcasts), lack of buffer space (reflected as a source quench), or unreachable host. Note however, in accordance with the ICMP protocol specification, ICMP messages are returned only for the first fragment of fragmented datagrams.

It is possible to disable the forwarding of datagrams by a host by patching the kernel variable *ipforwarding* to have value 0.

If a packet arrives and is destined for this machine, then IP must check to see if other fragments of the same datagram are being held. If this datagram is complete, then any previous fragments of it are discarded. If this is only a fragment of a datagram, it may yield a complete set of pieces for the datagram, in which case IP constructs the complete datagram and continues processing with that. If there is yet no complete set of pieces for this datagram, then all data thus far received is held (but only one copy of each data byte from the datagram) in hopes that the rest of the pieces of the fragmented datagram will arrive and we will be able to proceed. We allow IPFRAGTTL (currently 15 in <netinet/ip.h>) seconds for all the

fragments of a datagram to arrive, and discard partial fragments then if the datagram has not yet been completely assembled.

When we have a complete input datagram it is passed out to the appropriate protocol's input routine: either *tcp*(4P), *udp*(4P), *nd*(4P), *icmp*(4P) or a user process through a raw IP socket as described below.

Datagrams are output by the system-implemented protocols *tcp*(4P), *udp*(4P), *nd*(4P), and *icmp*(4P); as well as by packet forwarding operations and user processes through raw IP sockets. Output packets are normally subjected to routing as described in *routing*(4N). However, special processes such as the routing daemon *routed*(1M) occasionally use the SO_DONTROUTE socket option to make packets avoid the routing tables and go directly to the network interface with the network number which the packet is addressed to. This may be used to test the ability of the hardware to transmit and receive packets even when we believe that the hardware is broken and have therefore deleted it from the routing tables.

If there is no route to a destination address or if the SO_DONTROUTE option is given and there is no interface on the network specified by the destination address, then the IP output routine returns a ENETUNREACH error. (This and the other IP output errors are reflected back to user processes through the various protocols, which individually describe how errors are reported.)

In the (hopefully normal) case where there is a suitable route or network interface, the destination address is checked to see if it specifies a broadcast (address INADDR_ANY; see *inet*(4F)); if it does, and the hardware interface does not support broadcasts, then an EADDRNOTAVAIL is returned; if the caller is not the super-user then a EACCESS error will be returned. IP also does not allow broadcast messages to be fragmented, returning a EMSGSIZE error in this case.

If the datagram passes all these tests, and is small enough to be sent in one chunk, then the system calls the output routine for the particular hardware interface to transmit the packet. The interface may give an error indication, which is reflected to IP output's caller; see the documentation for the specific interface for a description of errors it may encounter. If a datagram is to be fragmented, it may have the IP_DF (don't fragment) flag set (although currently this can happen only for forwarded datagrams). If it does, then the datagram will be rejected (and result in an ICMP error datagram). If the system runs out of buffer space in fragmenting a datagram then a ENOBUFS error will be returned.

IP provides a space of 255 protocols. The known protocols are defined in <netinet/in.h>. The ICMP, TCP, UDP and ND protocols are processed internally by the system; others may be accessed through a raw socket by doing:

**s = socket(AF_INET, SOCK_RAW, IPPROTO_xxx);**

Datagrams sent from this socket will have the current host's address and the specified protocol number; the raw IP driver will construct an appropriate header. When IP datagrams are received for this protocol they are queued on the raw socket where they may be read with *recvfrom;* the source IP address is reflected in the received address.

**SEE ALSO**

send(2), recv(2), inet(4F)

Internet Protocol, RFC791, USC-ISI (Sun 800-1063-01)

**ERRORS**

One should be able to send and receive IP options.

Raw sockets should receive ICMP error packets relating to the protocol; currently such packets are simply discarded.

**NAME**

ip – Internet Protocol

**SYNOPSIS**

None; included by default with *inet*(7F).

**DESCRIPTION**

The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. It provides for transmitting blocks of data called "datagrams" from sources to destinations, where sources and destinations are hosts identified by fixed-length addresses. It also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks.

IP is specifically limited in scope. There are no mechanisms to augment end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols. IP can capitalize on the services of its supporting networks to provide various types and qualities of service.

IP is called on by host-to-host protocols, including *udp*(7P), a socket-socket datagram protocol. Other protocols may be layered on top of IP using the *raw* protocol facilities described here to receive and send datagrams with a specific IP protocol number. The IP protocol calls on local network drivers to carry the internet datagram to the next gateway or destination host.

When a datagram arrives at a UNIX system host, the system performs a checksum on the header of the datagram. If this fails, or if the datagram is unreasonably short or the header length specified in the datagram is not within range, then the datagram is dropped. Checksumming of Internet datagrams may be disabled for debugging purposes by patching the kernel variable *ipcksum* to have the value 0.

Next the system scans the IP options of the datagram. Options allowing for source routing (see *routed*(1M)) and also the collection of time stamps as a packet follows a particular route (for network monitoring and statistics gathering purposes) are handled; other options are ignored. Processing of source routing options may result in an UNREACH *icmp*(7P) message because the source routed host is not accessible.

After processing the options, IP checks to see if the current machine is the destination for the datagram. If not, then IP attempts to forward the datagram to the proper host. Before forwarding the datagram, IP decrements the time to live field of the datagram by IPTTLDEC seconds (currently 5 from <netinet/ip.h>), and discards the datagram if its lifetime has expired, sending an ICMP TIMXCEED error packet back to the source host. Similarly if the attempt to forward the datagram fails, then ICMP messages indicating an unreachable network, datagram too large, unreachable port (datagram would have required broadcasting on the target interface, and IP does not allow directed broadcasts), lack of buffer space (reflected as a source quench), or unreachable host. Note however, in accordance with the ICMP protocol specification, ICMP messages are returned only for the first fragment of fragmented datagrams.

It is possible to disable the forwarding of datagrams by a host by patching the kernel variable *ipforwarding* to have value 0.

If a packet arrives and is destined for this machine, then IP must check to see if other fragments of the same datagram are being held. If this datagram is complete, then any previous fragments of it are discarded. If this is only a fragment of a datagram, it may yield a complete set of pieces for the datagram, in which case IP constructs the complete datagram and continues processing with that. If there is yet no complete set of pieces for this datagram, then all data thus far received is held (but only one copy of each data byte from the datagram) in hopes that the rest of the pieces of the fragmented datagram will arrive and we will be able to proceed. We allow IPFRAGTTL (currently 15 in <netinet/ip.h>) seconds for all the fragments of a datagram to arrive, and discard partial fragments then if the datagram has not yet

been completely assembled.

When we have a complete input datagram it is passed out to the appropriate protocol's input routine: either *udp*(7P), *icmp*(7P), or a user process through a raw IP socket as described below.

Datagrams are output by the system-implemented protocols *udp*(7P) and *icmp*(7P) as well as by packet forwarding operations and user processes through raw IP sockets. Output packets are normally subjected to routing as described in *routed*(1M). However, special processes such as the routing daemon *routed*(1M) occasionally use the SO_DONTROUTE socket option to make packets avoid the routing tables and go directly to the network interface with the network number which the packet is addressed to. This may be used to test the ability of the hardware to transmit and receive packets even when we believe that the hardware is broken and have therefore deleted it from the routing tables.

If there is no route to a destination address or if the SO_DONTROUTE option is given and there is no interface on the network specified by the destination address, then the IP output routine returns a ENETUNREACH error. (This and the other IP output errors are reflected back to user processes through the various protocols, which individually describe how errors are reported.)

In the (hopefully normal) case where there is a suitable route or network interface, the destination address is checked to see if it specifies a broadcast (address INADDR_ANY; see *inet*(7F)); if it does, and the hardware interface does not support broadcasts, then an EADDRNOTAVAIL is returned; if the caller is not the super-user then a EACCESS error will be returned. IP also does not allow broadcast messages to be fragmented, returning a EMSGSIZE error in this case.

If the datagram passes all these tests, and is small enough to be sent in one chunk, then the system calls the output routine for the particular hardware interface to transmit the packet. The interface may give an error indication, which is reflected to IP output's caller; see the documentation for the specific interface for a description of errors it may encounter. If a datagram is to be fragmented, it may have the IP_DF (don't fragment) flag set (although currently this can happen only for forwarded datagrams). If it does, then the datagram will be rejected (and result in an ICMP error datagram). If the system runs out of buffer space in fragmenting a datagram then a ENOBUFS error will be returned.

IP provides a space of 255 protocols. The known protocols are defined in <netinet/in.h>. The ICMP and UDP protocols are processed internally by the system; others may be accessed through a raw socket by doing:

**s = socket(AF_INET, SOCK_RAW, IPPROTO_xxx);**

Datagrams sent from this socket will have the current host's address and the specified protocol number; the raw IP driver will construct an appropriate header. When IP datagrams are received for this protocol they are queued on the raw socket where they may be read with *recvfrom;* the source IP address is reflected in the received address.

**SEE ALSO**

*send*(2), *recv*(2), *inet*(7F)

Internet Protocol, RFC791, USC-ISI (Sun 800-1063-01)

**ERRORS**

One should be able to send and receive IP options.

Raw sockets should receive ICMP error packets relating to the protocol; currently such packets are simply discarded.

**NAME**

kbd – RS2030 Display Keyboard Driver

**SYNOPSIS**

**display keyboard on RS2030 IOP**

**DESCRIPTION**

The *kbd* driver supports input from the RS2030 display keyboard and control functions for the keyboard lights and the system buzzer.

This driver serves as both a terminal driver and a specialized driver for more direct control of the system keyboard.

*/dev/ttykeybd* represents the terminal driver. This device function as a standard serial streams device, using the keyboard for input and the raw display character painting functions for output to the system display. This driver converts keyboard input codes to ASCII characters.

*/dev/keybd* represents the specialized keyboard control driver. This returns keyboard input codes unchanged. They may be timestamped by using the KTCSETTIMESTAMP *ioctl*. The driver also supports several other specialized functions for using the system buzzer and keyboard lights (see below). Output to this driver go to where the system console is linked.

The following *ioctl* calls apply only to the */dev/keybd:*

| | |
|---|---|
| KTCSETTIMESTAMP | Turns on timestamping of all input from the keyboard. Each character received from the keyboard is presented in a timestamp structure. These are composed sync characters, the keyboard character, and the time in HZ since boot. The structure is defined in */usr/include/sys/kbd_ioctl.h.* |
| KTCCLRTIMESTAMP | Turns off keyboard input timestamping. |
| KTCPRGMBUZZER | Allows sending a *buzzer* structure to the system to program the system buzzer. This structure is defined in */usr/include/sys/buzzer.h.* |
| KTCRINGBELL | Rings the buzzer to emulate a terminal bell (control-g). This function uses only the command field of the *ioctl.* |
| KTCSETLIGHTS | Turns on and off keyboard lights. The lights are turned on based on the bits set in the parameter passed. The values for each light are in */usr/include/sys/kbd_ioctl.h.* The old state of the lights are not maintained. To just turn one light on/off, you must KTCGETLIGHTS and or/and the new light state into that value for the KTCSETLIGHTS. |
| KTCGETLIGHTS | Returns last value of KTCSETLIGHTS. |
| KTCWRTCOLOR | Write the color map. The following structure defines which color map registers are written: |

```
struct colorm {
        unsigned short cmstart;         /* starting map register number */
        unsigned short cmcount;                 /* number of registers to write */
        unsigned char cmap[3*256];      /* register values */
};
```

The map is written the next valid vertical retrace interrupt.

**FILES**

*/dev/ttykeybd*
*/dev/keydb*

**SEE ALSO**

*tty(7)*
*iop(7)*

**NAME**

la – AMD 7990 Ethernet interface

**SYNOPSIS**

FLAG PREFIXSOFT #DEV DEPENDENCIES – if_la – – bsd,iop

**DESCRIPTION**

The *la* interface provides access to a 10 Mb/s Ethernet network through an AMD 7990 lance chip.

The host's network address is specified at boot time with an SIOCSIFADDR ioctl. The station address is discovered by reading an on-board prom. No packets are sent or accepted until a network address is supplied. The *la* interface employs the address resolution protocol described in *arp(4P)* to dynamically map between Internet and Ethernet addresses on the local network.

The interface handles both Internet and NS protocol families. The use of trailers is negotiated with ARP. However, this negotiation may be disabled by setting the IFF_NOTRAILERS flag with an SIOCSIFFLAGS ioctl.

**DIAGNOSTICS**

IOP: lance0 %s is available. The '%s' is replaced by a six byte hardware address and is a normal part of the configuration process.

**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. As with other network interfaces, the loopback interface must have network addresses assigned for each address family with which it is to be used. These addresses may be set or changed with the SIOCSIFADDR ioctl. The loopback interface should be the last interface configured, as protocols may use the order of configuration as an indication of priority. The loopback should **never** be configured first unless no hardware interfaces exist.

**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(4N), inet(4F), ns(4F)

**ERRORS**

Previous versions of the system enabled the loopback interface automatically, using a nonstandard Internet address (127.1). Use of that address is now discouraged; a reserved host address for the local network should be used instead.

**NAME**

log – interface to STREAMS error logging and event tracing

**DESCRIPTION**

*log* is a STREAMS software device driver that provides an interface for the STREAMS error logging and event tracing processes (*strerr*(1M), *strace*(1M)). *log* presents two separate interfaces: a function call interface in the kernel through which STREAMS drivers and modules submit *log* messages; and a subset of *ioctl*(2) system calls and STREAMS messages for interaction with a user level error logger, a trace logger, or processes that need to submit their own *log* messages.

**Kernel Interface**

*log* messages are generated within the kernel by calls to the function *strlog*:

```
strlog(mid, sid, level, flags, fmt, arg1, ...)
short mid, sid;
char level;
ushort flags;
char *fmt;
unsigned arg1;
```

Required definitions are contained in **<sys/strlog.h>** and **<sys/log.h>**. *mid* is the STREAMS module id number for the module or driver submitting the *log* message. *sid* is an internal sub-id number usually used to identify a particular minor device of a driver. *level* is a tracing level that allows for selective screening out of low priority messages from the tracer. *flags* are any combination of SL_ERROR (the message is for the error logger), SL_TRACE (the message is for the tracer), SL_FATAL (advisory notification of a fatal error), and SL_NOTIFY (request that a copy of the message be mailed to the system administrator). *fmt* is a *printf(3S)* style format string, except that %s, %e, %E, %g, and %G conversion specifications are not handled. Up to NLOGARGS (currently 3) numeric or character arguments can be provided.

**User Interface**

*log* is opened via the clone interface, **/dev/log**. Each open of **/dev/log** obtains a separate *stream* to *log*. In order to receive *log* messages, a process must first notify *log* whether it is an error logger or trace logger via a STREAMS I_STR *ioctl* call (see below). For the error logger, the I_STR *ioctl* has an ic_cmd field of I_ERRLOG, with no accompanying data. For the trace logger, the *ioctl* has an ic_cmd field of I_TRCLOG, and must be accompanied by a data buffer containing an array of one or more struct trace_ids elements. Each trace_ids structure specifies an *mid*, *sid*, and *level* from which message will be accepted. *strlog* will accept messages whose *mid* and *sid* exactly match those in the trace_ids structure, and whose level is less than or equal to the level given in the trace_ids structure. A value of -1 in any of the fields of the trace_ids structure indicates that any value is accepted for that field.

At most one trace logger and one error logger can be active at a time. Once the logger process has identified itself via the *ioctl* call, *log* will begin sending up messages subject to the restrictions noted above. These messages are obtained via the *getmsg(2)* system call. The control part of this message contains a log_ctl structure, which specifies the *mid*, *sid*, *level*, *flags*, time in ticks since boot that the message was submitted, the corresponding time in seconds since Jan. 1, 1970, and a sequence number. The time in seconds since 1970 is provided so that the date and time of the message can be easily computed, and the time in ticks since boot is provided so that the relative timing of *log* messages can be determined.

Different sequence numbers are maintained for the error and trace logging *streams*, and are provided so that gaps in the sequence of messages can be determined (during times of high message traffic some messages may not be delivered by the logger to avoid hogging system resources). The data part of the message contains the unexpanded text of the format string

(null terminated), followed by NLOGARGS words for the arguments to the format string, aligned on the first word boundary following the format string.

A process may also send a message of the same structure to *log*, even if it is not an error or trace logger. The only fields of the log_ctl structure in the control part of the message that are accepted are the level and flags fields; all other fields are filled in by *log* before being forwarded to the appropriate logger. The data portion must contain a null terminated format string, and any arguments (up to NLOGARGS) must be packed one word each, on the next word boundary following the end of the format string.

Attempting to issue an I_TRCLOG or I_ERRLOG when a logging process of the given type already exists will result in the error ENXIO being returned. Similarly, ENXIO is returned for I_TRCLOG *ioctls* without any trace_ids structures, or for any unrecognized I_STR *ioctl* calls. Incorrectly formatted *log* messages sent to the driver by a user process are silently ignored (no error results).

**EXAMPLES**

Example of I_ERRLOG notification.

```
struct strioctl ioc;

ioc.ic_cmd = I_ERRLOG;
ioc.ic_timout = 0;              /* default timeout (15 secs.) */
ioc.ic_len = 0;
ioc.ic_dp = NULL;

ioctl(log, I_STR, &ioc);
```

Example of I_TRCLOG notification.

```
struct trace_ids tid[2];

tid[0].ti_mid = 2;
tid[0].ti_sid = 0;
tid[0].ti_level = 1;

tid[1].ti_mid = 1002;
tid[1].ti_sid = -1;         /* any sub-id will be allowed */
tid[1].ti_level = -1;       /* any level will be allowed */

ioc.ic_cmd = I_TRCLOG;
ioc.ic_timout = 0;
ioc.ic_len = 2 * sizeof(struct trace_ids);
ioc.ic_dp = (char *)tid;

ioctl(log, I_STR, &ioc);
```

Example of submitting a *log* message (no arguments).

```
struct strbuf ctl, dat;
struct log_ctl lc;
char *message = "Don't forget to pick up some milk on the way home";

ctl.len = ctl.maxlen = sizeof(lc);
ctl.buf = (char *)&lc;
```

```
dat.len = dat.maxlen = strlen(message);
dat.buf = message;

lc.level = 0;
lc.flags = SL_ERROR|SL_NOTIFY;

putmsg(log, &ctl, &dat, 0);
```

**FILES**

/dev/log, <sys/log.h>, <sys/strlog.h>

**SEE ALSO**

strace(1M), strerr(1M), clone(7).
intro(2), getmsg(2), putmsg(2) in the *Programmer's Reference Manual*.
*STREAMS Programmer's Guide*.

**NAME**

lp – parallel port/line printer interface

**SYNOPSIS**

**VECTOR: module=lp vector=0x12 ipl=0 unit=0 base=000**

**DESCRIPTION**

The parallel port is a Centronics compatible interface used for output to a line printer.

The raw device sends all characters out the port unmodified. The canonical device attempts to manage column and line position, handling backspaces, carriage returns, linefeeds and formfeeds for "dumb" printers. The capitalize device converts lowercase characters to uppercase and prints certain special characters as other characters with overstrikes, as well as canonizing, for "really dumb" printers which cannot print the entire ascii set.

**FILES**

| | |
|---|---|
| /dev/rlp0 | raw device |
| /dev/lp0 | canonical device |
| /dev/Lp0 | capitalize device |

**DIAGNOSTICS**

| | |
|---|---|
| lp0: not ready | Printer not ready. |
| lp0: not cabled correctly | Hardware detected no cable. |
| lp0: out of paper | Printer out of paper. |
| lp0: not selected (offline) | Printer not online. |
| lp0: not responding | Printer not responding with acknowledge. |
| lp0: cannot alloc mem | Driver could not allocate canonical buffer memory during open. System error. |
| lp0: interrupt from non-existent device | Interrupt routine was called with an illegal port number. System error. |
| lp0: Stray Interrupt | An unexpected interrupt was received while device was not active. System error. |
| lp0: bad command | An illegal command was given to the hardware. System error. |
| lp0: cannot initialize | Parallel port controller could not be initialized during boot. System error. |
| lp0: device busy: command rejected | Print command was given while port was busy. System error. |

**NAME**

lp – RS2030/RC2030 IOP Line Printer Parallel Port Driver

**SYNOPSIS**

**line printer parallel port on RC2030 IOP**

**DESCRIPTION**

The *lp* driver supports the parallel port on the RS2030/RC2030 IOP for line printers.

The devices number is used to determine which of three modes that the devices supports, as well as the number of columns on the device. The device number has the following format:

| | | |
|---|---|---|
| bit 0 | unit | (only 0 allowed) |
| bits 1,2 mode | | (RAW=2,CAPS=4,CANONICAL=0) |
| bits 3-7 columns | | (default 80,maximum=1000) |

In *RAW* mode all characters are sent directly out the port.

In *CAPS* mode, the letters a-z are converted to capitals and the following characters are printed converted as descirbed and over-struck with a '-':

{ converted to (
} converted to )
` converted to \
| converted to !
~ converted to ^

All other characters are unchanged.

*CANONICAL* supports \f (forms feed), \r (carriage return only), \n (carriage return/line feed), and \b (backspace). Line wrap is supported when the number of columns is exceeded.

**FILES**

*/dev/lp*

**SEE ALSO**

*iop(7)*

**NAME**

mailaddr – mail addressing description

**DESCRIPTION**

Mail addresses are based on the ARPANET protocol listed at the end of this manual page. These addresses are in the general format

user@domain

where a domain is a hierarchical dot separated list of subdomains. For example, the address

eric@monet.Berkeley.ARPA

is normally interpreted from right to left: the message should go to the ARPA name tables (which do not correspond exactly to the physical ARPANET), then to the Berkeley gateway, after which it should go to the local host monet. When the message reaches monet it is delivered to the user "eric".

Unlike some other forms of addressing, this does not imply any routing. Thus, although this address is specified as an ARPA address, it might travel by an alternate route if that were more convenient or efficient. For example, at Berkeley the associated message would probably go directly to monet over the Ethernet rather than going via the Berkeley ARPANET gateway.

**Abbreviation.**

Under certain circumstances it may not be necessary to type the entire domain name. In general anything following the first dot may be omitted if it is the same as the domain from which you are sending the message. For example, a user on "calder.Berkeley.ARPA" could send to "eric@monet" without adding the ".Berkeley.ARPA" since it is the same on both sending and receiving hosts.

Certain other abbreviations may be permitted as special cases. For example, at Berkeley ARPANET hosts can be referenced without adding the ".ARPA" as long as their names do not conflict with a local host name.

**Compatibility.**

Certain old address formats are converted to the new format to provide compatibility with the previous mail system. In particular,

host:user

is converted to

user@host

to be consistent with the *rcp*(1C) command.

Also, the syntax:

host!user

is converted to:

user@host.UUCP

This is normally converted back to the "host!user" form before being sent on for compatibility with older UUCP hosts.

The current implementation is not able to route messages automatically through the UUCP network. Until that time you must explicitly tell the mail system which hosts to send your message through to get to your final destination.

**Case Distinctions.**

Domain names (i.e., anything after the "@" sign) may be given in any mixture of upper and lower case with the exception of UUCP hostnames. Most hosts accept any combination of case in user names, with the notable exception of MULTICS sites.

**Differences with ARPA Protocols.**

Although the UNIX addressing scheme is based on the ARPA mail addressing protocols, there are some significant differences.

At the time of this writing DARPA is converting to real domains. The following rules may be useful:

- The syntax "user@host.ARPA" is being split up into "user@host.COM", "user@host.GOV", and "user@host.EDU" for commercial, government, and educational institutions respectively.

- The syntax "user@host" (with no dots) has traditionally referred to the ARPANET. In the future this semantic will not be continued – instead, the host will be assumed to be in your organization. You should start using one of the syntaxes above.

- Host names of the form "ORG-NAME" (e.g., MIT-MC or CMU-CS-A) will be changing to "NAME.ORG.XXX" (where 'XXX' is COM, GOV, or EDU). For example, MIT-MC will change to MC.MIT.EDU. In some cases names will be split apart even if they do not have dashes. For example, USC-ISIF will probably change to F.ISI.USC.EDU.

**Route-addrs.**

Under some circumstances it may be necessary to route a message through several hosts to get it to the final destination. Normally this routing is done automatically, but sometimes it is desirable to route the message manually. Addresses which show these relays are termed "route-addrs." These use the syntax:

<@hosta,@hostb:user@hostc>

This specifies that the message should be sent to hosta, from there to hostb, and finally to hostc. This path is forced even if there is a more efficient path to hostc.

Route-addrs occur frequently on return addresses, since these are generally augmented by the software at each host. It is generally possible to ignore all but the "user@host" part of the address to determine the actual sender.

**Postmaster.**

Every site is required to have a user or user alias designated "postmaster" to which problems with the mail system may be addressed.

**Other Networks.**

Some other networks can be reached by giving the name of the network as the last component of the domain. *This is not a standard feature* and may not be supported at all sites. For example, messages to CSNET or BITNET sites can often be sent to "user@host.CSNET" or "user@host.BITNET" respectively.

**BERKELEY**

The following comments apply only to the Berkeley environment.

**What's My Address?**

If you are on a local machine, say monet, your address is

yourname@monet.Berkeley.ARPA

However, since most of the world does not have the new software in place yet, you will have to give correspondents slightly different addresses. From the ARPANET, your address would be:

yourname%monet@Berkeley.ARPA

From UUCP, your address would be:

ucbvax!yourname%monet

**Computer Center.**

The Berkeley Computer Center is in a subdomain of Berkeley. Messages to the computer center should be addressed to:

user%host.CC@Berkeley.ARPA

The alternate syntax:

user@host.CC

may be used if the message is sent from inside Berkeley.

For the time being Computer Center hosts are known within the Berkeley domain, i.e., the ".CC" is optional. However, it is likely that this situation will change with time as both the Computer Science department and the Computer Center grow.

**ERRORA**

The RFC822 group syntax ("group:user1,user2,user3;") is not supported except in the special case of "group:;" because of a conflict with old berknet-style addresses.

Route-Address syntax is grotty.

UUCP- and ARPANET-style addresses do not coexist politely.

**SEE ALSO**

mail(1), sendmail(1M); Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages,* RFC822.

**NAME**

mem, kmem – main memory

**DESCRIPTION**

*mem* is a special file that is an image of the main memory of the computer. It may be used, for example, to examine (and even to patch) the system.

Byte addresses in *mem* are interpreted as physical memory addresses. References to non-existent locations cause errors to be returned.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

**FILES**

/dev/mem
/dev/kmem

**NAME**

mt – mag tape interface

**DESCRIPTION**

The files **/dev/mt/h*** and **/dev/rmt/h*** refer to mag tape controllers (1/2" 9-track) and associated tape drives. The ioctls are the same as denoted in **mtio(7)**.

The minor devices for the mag tape are defined as follows:

| | |
|---|---|
| 0 | Rewind |
| 4 | No Rewind |

**FILES**

/dev/mt/h*
/dev/rmt/h*

**SEE ALSO**

mtio(7), st(7)

**NAME**

mtio – UNIX tape interface

**DESCRIPTION**

The files */dev/[r]mt/** refer to the UNIX tape drives, which may be on the VMEBUS using the ISI QIC-2 streaming tape *st(7)* or the 1/2 inch 9 track tape *mt(7)*.

Many ioctl operations are available on raw magnetic tape. The following definitions are from *<sys/mtio.h>*:

```
/*
 * Structures and definitions for mag tape io control commands
 */

/* structure for MTIOCTOP - mag tape op command */
struct  mtop  {
        short   mt_op;          /* operations defined below */
        daddr_t mt_count;       /* how many of them */
};

/* operations */
#define MTWEOF      0       /* write an end-of-file record */
#define MTFSF       1       /* forward space file */
#define MTBSF       2       /* backward space file */
#define MTFSR       3       /* forward space record */
#define MTBSR       4       /* backward space record */
#define MTREW       5       /* rewind */
#define MTOFFL      6       /* rewind and put the drive offline */
#define MTNOP       7       /* no operation, sets status only */
#define MTCACHE     8       /* enable controller cache */
#define MTNOCACHE   9       /* disable controller cache */
#define MTRET       10      /* retention operation */
#define MTRST       11      /* reset operation */

/* structure for MTIOCGET - mag tape get status command */

struct  mtget  {
        short   mt_type;        /* type of magtape device */
/* the following two registers are grossly device dependent */
        short   mt_dsreg;       /* "drive status" register */
        short   mt_erreg;       /* "error" register */
/* end device-dependent registers */
        short   mt_resid;       /* residual count */
/* the following two are not yet implemented */
        daddr_t mt_fileno;      /* file number of current position */
        daddr_t mt_blkno;       /* block number of current position */
/* end not yet implemented */
};

/*
 * Constants for mt_type byte.  These are the same
 * for other controllers compatible with the types listed.
 */
#define MT_ISTS             0x01            /* TS-11 */
```

```
#define MT_ISHT         0x02        /* TM03 Massbus: TE16, TU45, TU77 */
#define MT_ISTM         0x03        /* TM11/TE10 Unibus */
#define MT_ISMT         0x04        /* TM78/TU78 Massbus */
#define MT_ISUT         0x05        /* SI TU-45 emulation on Unibus */
#define MT_ISCPC    0x06        /* SUN */
#define MT_ISAR         0x07        /* SUN */
#define MT_ISTMSCP  0x08        /* DEC TMSCP protocol (TU81, TK50) */
#define MT_ISQIC    0x09        /* ISI ts11 qic-2 tape controller */

/* mag tape io control commands */
#define MTIOCTOP    _IOW(m, 1, struct mtop)         /* do a mag tape op */
#define MTIOCGET    _IOR(m, 2, struct mtget)    /* get tape status */
#define MTIOCIEOT   _IO(m, 3)                   /* ignore EOT error */
#define MTIOCEEOT   _IO(m, 4)                   /* enable EOT error */

#ifndef KERNEL
#define DEFTAPE     "/dev/rmt12"
#endif
```

Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

**FILES**

/dev/mt/*
/dev/rmt/*

**SEE ALSO**

mt(7), tar(1), cpio(1), st(7)

**ERRORS**

The status should be returned in a device independent format.

The special file naming should be redone in a more consistent and understandable manner.

**NAME**

null – data sink

**DESCRIPTION**

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

**FILES**

/dev/null

**NAME**

prf – operating system profiler

**DESCRIPTION**

The special file */dev/prf* provides access to activity information in the operating system. Writing the file loads the measurement facility with text addresses to be monitored. Reading the file returns these addresses and a set of counters indicative of activity between adjacent text addresses.

The recording mechanism is driven by the system clock and samples the program counter at line frequency. Samples that catch the operating system are matched against the stored text addresses and increment corresponding counters for later processing.

The file */dev/prf* is a pseudo-device with no associated hardware.

**FILES**

/dev/prf

**SEE ALSO**

profiler(1M).

**NAME**

pty – pseudo terminal driver

**SYNOPSIS**

**pseudo-device pty [ count ]**

**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 *tty*(7). However, whereas all other devices which provide the interface described in *tty*(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 an optional "count" is given in the specification, that number of pseudo terminal pairs are configured; the default count is 64.

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.

  While this mode is in use, the presence of control status information to be read from the master side may be detected by a *select* for exceptional conditions.

  This mode is used by *rlogin*(1C) and *rlogind*(1M) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output; it can be used by other similar programs.

TIOCUCNTL
: Enable/disable a mode that allows a small number of simple user *ioctl* commands to be passed through the pseudo-terminal, using a protocol similar to that of TIOCPKT. The TIOCUCNTL and TIOCPKT modes are mutually exclusive. This mode is enabled from the master side of a pseudo terminal by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. Each subsequent *read* from the master side will return data written on the slave part of the pseudo terminal preceded by a zero byte, or a single byte reflecting a user control operation on the slave side. A user control command consists of a special *ioctl* operation with no data; the command is given as UIOCCMD(*n*), where *n* is a number in the range 1-255. The operation value *n* will be received as a single byte on the next *read* from the master side. The *ioctl* UIOCCMD(0) is a no-op that may be used to probe for the existence of this facility. As with TIOCPKT mode, command operations may be detected with a *select* for exceptional conditions.

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 editing in a window manager, or whenever flow controlled input is required.

PTCSETCON
: A mode for the slave side of a psuedo terminal, for use by the super-user only. This mode causes the kernel output to the console to be linked to the output of this slave psuedo terminal. The master psuedo teminal may do what ever is appropriate for kernel output. On last close of this slave psuedo terminal, a PTCCLRCON is performed. Note: this is used by xterm when the -C option is used. Using this option requires that the user include */usr/include/sys/termio.h* and */usr/include/sys/pty_ioctl.h*. Only the cmd field is used for this *ioctl*.

PTCCLRCON
: A mode for the slave side of a psuedo terminal, for use by the super-user only. This mode causes the kernel output to the console to be unlinked from the output of this slave psuedo terminal. Using this option requires that the user include */usr/include/sys/termio.h* and */usr/include/sys/pty_ioctl.h*. This function is effectively executed on the last close of the master psuedo terminal that is reading the console output. Only the cmd field is used for this *ioctl*.

**FILES**

| | |
|---|---|
| /dev/pty[p-s][0-9a-f] | master pseudo terminals |
| /dev/tty[p-s][0-9a-f] | slave pseudo terminals |

**DIAGNOSTICS**

None.

**NAME**

qt – SCSI QIC-100 tape interface

**SYNOPSIS**

**VECTOR: module=sha vector=0x8 ipl=2 unit=1 base=010**

**DESCRIPTION**

The QIC-100 SCSI tape is a 3-1/2 inch random access tape used for backups, software distribution, and file transfer.

In raw I/O, buffers should be page aligned for best performance and I/O counts should be a multiple of 16384 bytes (a pair of overlapped tape blocks). Likewise *seek* calls should specify a multiple of 16384 bytes.

ISI QIC-100 tapes contain a 16 kilobyte directory in the first two blocks which allows multiple files on tape. The files can contain blocks of variable length, but any particular file must contain only one block length (except for the last block which may be shorter or longer). Ioctl calls to space forward (reverse) files or records use the directory information to go directly to the correct block on the random access tape.

Since the tape format works with 8k blocks (16k pairs of overlapped blocks in the case of 1:1 interleave), accesses which are not in multiples of 16k are buffered by the driver. In any event, the directory is not updated until the device is closed. Therefore, tapes should never be removed from the drive while the device is open.

**FILES**

/dev/mt/ctape0 block rewind device
/dev/mt/ctape4 block non-rewind device
/dev/rmt/m0 raw rewind device
/dev/rmt/m4 raw non-rewind device

**SEE ALSO**

mt(1)

**DIAGNOSTICS**

The following diagnostics are printed by the *qt* driver:

qt%d mode sense failed
: Although the tape drive was present, the driver was unsuccessful in requesting drive specific information.

qt%d: no cartridge or tape not formatted
: The driver was unable to read the first block from the tape, either because no cartridge was inserted, the tape was unformatted, or the drive was defective.

qt%d: new tape: creating directory
: The first block of the tape did not contain a valid directory. The driver creates a new empty directory. This directory is not written to the tape unless a write call is executed.

qt%d: tape is write protected
: Self explanatory.

qt%d: record size changed twice
: The qictp driver handles fixed blocksize files, with the exception that the last block in a file may be a different size. All attempts to write to a tape file after a short (or long) block is written will be rejected with this message.

qt%d: reassigning block %d (0x%x)...done

qt%d: reassigning block %d (0x%x)...failed
: If the reassigning device is used (GT_NO_REASSIGN not set) and an unrecoverable medium write error occurs, the driver will attempt to reassign the failing logical block and retry the write. Reassignment may fail if the alternate block is already in use.

qt%d: tape changed or drive reset
: The drive asserted unit attention, indicating a reset, power up, or changed tape cartridge. To prevent destruction of data, no more accesses are permitted until the device is closed and reopened.

qt%d: tape blocksize wrong
: Attached drive is apparently wrong type, and will not work with driver.

qt%d: %s: code %x : reissued
: Various SCSI drive errors: see disk drive manual or call Technical Support for detailed explanation. When an error has occurred the driver will retry the command up to 3 times.

**SEE ALSO**

tpd(5spp)

**NAME**

sa – devices administered by System Administration

**DESCRIPTION**

The files in the directories **/dev/SA** (for block devices) and the **/dev/rSA** (for raw devices) are used by System Administration to access the devices on which it operates. For devices that support more than one partition (like disks) the **/dev/(r)SA** entry is linked to the partition that spans the entire device. Not all **/dev/(r)SA** entries are used by all System Administration commands.

**FILES**

/dev/SA
/dev/rSA

**SEE ALSO**

sysadm(1) in the *User's Reference Manual*.

spp – Xerox Sequenced Packet Protocol

**SYNOPSIS**

**#include <sys/socket.h>**
**#include <netns/ns.h>**
**s = socket(AF_NS, SOCK_STREAM, 0);**

**#include <netns/sp.h>**
**s = socket(AF_NS, SOCK_SEQPACKET, 0);**

**DESCRIPTION**

The SPP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. SPP uses the standard NS(tm) address formats.

Sockets utilizing the SPP protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default SPP sockets are created active; to create a passive socket the *listen*(2) system call must be used after binding the socket with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) 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 NS address of all zeroes must be bound. The SPP 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 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.

If the SOCK_SEQPACKET socket type is specified, each packet received has the actual 12 byte sequenced packet header left for the user to inspect:

```
struct sphdr {
        u_char          sp_cc;          /* connection control */
#define SP_EM 0x10                      /* end of message */
        u_char          sp_dt;          /* datastream type */
        u_short         sp_sid;
        u_short         sp_did;
        u_short         sp_seq;
        u_short         sp_ack;
        u_short         sp_alo;
};
```

This facilitates the implementation of higher level Xerox protocols which make use of the data stream type field and the end of message bit. Conversely, the user is required to supply a 12 byte header, the only part of which inspected is the data stream type and end of message fields.

For either socket type, packets received with the Attention bit sent are interpreted as out of band data. Data sent with send(..., ..., ..., MSG_OOB) cause the attention bit to be set.

**DIAGNOSTICS**

A socket operation may fail with one of the following errors returned:

| | |
|---|---|
| [EISCONN] | when trying to establish a connection on a socket which already has one; |
| [ENOBUFS] | when the system runs out of memory for an internal data structure; |

[ETIMEDOUT] when a connection was dropped due to excessive retransmissions;

[ECONNRESET] when the remote peer forces the connection to be closed;

[ECONNREFUSED] when the remote peer actively refuses connection establishment (usually because no process is listening to the port);

[EADDRINUSE] when an attempt is made to create a socket with a port which has already been allocated;

[EADDRNOTAVAIL] when an attempt is made to create a socket with a network address for which no network interface exists.

**SOCKET OPTIONS**

SO_DEFAULT_HEADERS
when set, this determines the data stream type and whether the end of message bit is to be set on every ensuing packet.

SO_MTU This specifies the maximum ammount of user data in a single packet. The default is 576 bytes - sizeof(struct spidp). This quantity affects windowing – increasing it without increasing the amount of buffering in the socket will lower the number of unread packets accepted. Anything larger than the default will not be forwarded by a bona fide XEROX product internetwork router. The data argument for the setsockopt call must be an unsigned short.

**SEE ALSO**

*intro*(7N)

**ERRORS**

There should be some way to reflect record boundaries in a stream. For stream mode, there should be an option to get the data stream type of the record the user process is about to receive.

**NAME**

streamio – STREAMS ioctl commands

**SYNOPSIS**

**#include <stropts.h>**
**int ioctl (fildes, command, arg)**
**int fildes, command;**

**DESCRIPTION**

STREAMS [see *intro*(2)] ioctl commands are a subset of *ioctl*(2) system calls which perform a variety of control functions on *streams*. The arguments *command* and *arg* are passed to the file designated by *fildes* and are interpreted by the *stream head*. Certain combinations of these arguments may be passed to a module or driver in the *stream*.

*fildes* is an open file descriptor that refers to a *stream*. *command* determines the control function to be performed as described below. *arg* represents additional information that is needed by this command. The type of *arg* depends upon the command, but it is generally an integer or a pointer to a *command*-specific data structure. Since these STREAMS commands are a subset of *ioctl*, they are subject to the errors described there. In addition to those errors, the call will fail with *errno* set to EINVAL, without processing a control function, if the *stream* referenced by *fildes* is linked below a multiplexor, or if *command* is not a valid value for a *stream*. Also, as described in *ioctl*, STREAMS modules and drivers can detect errors. In this case, the module or driver sends an error message to the *stream head* containing an error value. This causes subsequent system calls to fail with *errno* set to this value.

**COMMAND FUNCTIONS**

The following *ioctl* commands, with error values indicated, are applicable to all STREAMS files:

| | |
|---|---|
| I_PUSH | Pushes the module whose name is pointed to by *arg* onto the top of the current *stream*, just below the *stream head*. It then calls the open routine of the newly-pushed module. On failure, *errno* is set to one of the following values: |
| [EINVAL] | Invalid module name. |
| [EFAULT] | *arg* points outside the allocated address space. |
| [ENXIO] | Open routine of new module failed. |
| [ENXIO] | Hangup received on *fildes*. |
| I_POP | Removes the module just below the *stream head* of the *stream* pointed to by *fildes*. *arg* should be 0 in an I_POP request. On failure, *errno* is set to one of the following values: |
| [EINVAL] | No module present in the *stream*. |
| [ENXIO] | Hangup received on *fildes*. |
| I_LOOK | Retrieves the name of the module just below the *stream head* of the *stream* pointed to by *fildes*, and places it in a null terminated character string pointed at by *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long. An "#include <sys/conf.h>" declaration is required. On failure, *errno* is set to one of the following values: |
| [EFAULT] | *arg* points outside the allocated address space. |
| [EINVAL] | No module present in *stream*. |
| I_FLUSH | This request flushes all input and/or output queues, depending on the value of *arg*. Legal *arg* values are: |

| | |
|---|---|
| FLUSHR | Flush read queues. |
| FLUSHW | Flush write queues. |
| FLUSHRW | Flush read and write queues. |
| | On failure, *errno* is set to one of the following values: |
| [EAGAIN] | Unable to allocate buffers for flush message. |
| [EINVAL] | Invalid *arg* value. |
| [ENXIO] | Hangup received on *fildes*. |
| I_SETSIG | Informs the *stream head* that the user wishes the kernel to issue the SIGPOLL signal [see *signal*(2) and *sigset*(2)] when a particular event has occurred on the *stream* associated with *fildes*. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the user should be signaled. It is the bitwise-OR of any combination of the following constants: |
| S_INPUT | A non-priority message has arrived on a *stream head* read queue, and no other messages existed on that queue before this message was placed there. This is set even if the message is of zero length. |
| S_HIPRI | A priority message is present on the *stream head* read queue. This is set even if the message is of zero length. |
| S_OUTPUT | The write queue just below the *stream head* is no longer full. This notifies the user that there is room on the queue for sending (or writing) data downstream. |
| S_MSG | A STREAMS signal message that contains the SIGPOLL signal has reached the front of the *stream head* read queue. |
| | A user process may choose to be signaled only of priority messages by setting the *arg* bitmask to the value S_HIPRI. |
| | Processes that wish to receive SIGPOLL signals must explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same Stream, each process will be signaled when the event occurs. |
| | If the value of *arg* is zero, the calling process will be unregistered and will not receive further SIGPOLL signals. On failure, *errno* is set to one of the following values: |
| [EINVAL] | *arg* value is invalid or *arg* is zero and process is not registered to receive the SIGPOLL signal. |
| [EAGAIN] | Allocation of a data structure to store the signal request failed. |
| I_GETSIG | Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask pointed to by *arg*, where the events are those specified in the description of I_SETSIG above. On failure, *errno* is set to one of the following values: |
| [EINVAL] | Process not registered to receive the SIGPOLL signal. |
| [EFAULT] | *arg* points outside the allocated address space. |
| I_FIND | This request compares the names of all modules currently present in the *stream* to the name pointed to by *arg*, and returns 1 if the named module is present in the *stream*. It returns 0 if the named module is not present. On failure, *errno* is set to one of the following values: |

[EFAULT] *arg* points outside the allocated address space.

[EINVAL] *arg* does not contain a valid module name.

I_PEEK This request allows a user to retrieve the information in the first message on the *stream head* read queue without taking the message off the queue. *arg* points to a *strpeek* structure which contains the following members:

```
struct strbuf   ctlbuf;
struct strbuf   databuf;
long            flags;
```

The *maxlen* field in the *ctlbuf* and *databuf strbuf* structures [see *getmsg*(2)] must be set to the number of bytes of control information and/or data information, respectively, to retrieve. If the user sets *flags* to RS_HIPRI, I_PEEK will only look for a priority message on the *stream head* read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the *stream head* read queue, or if the RS_HIPRI flag was set in *flags* and a priority message was not present on the *stream head* read queue. It does not wait for a message to arrive. On return, *ctlbuf* specifies information in the control buffer, *databuf* specifies information in the data buffer, and *flags* contains the value 0 or RS_HIPRI. On failure, *errno* is set to the following value:

[EFAULT] *arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space.

I_SRDOPT Sets the read mode using the value of the argument *arg*. Legal *arg* values are:

RNORM Byte-stream mode, the default.

RMSGD Message-discard mode.

RMSGN Message-nondiscard mode.

Read modes are described in *read*(2). On failure, *errno* is set to the following value:

[EINVAL] *arg* is not one of the above legal values.

I_GRDOPT Returns the current read mode setting in an *int* pointed to by the argument *arg*. Read modes are described in *read*(2). On failure, *errno* is set to the following value:

[EFAULT] *arg* points outside the allocated address space.

I_NREAD Counts the number of data bytes in data blocks in the first message on the *stream head* read queue, and places this value in the location pointed to by *arg*. The return value for the command is the number of messages on the *stream head* read queue. For example, if zero is returned in *arg*, but the *ioctl* return value is greater than zero, this indicates that a zero-length message is next on the queue. On failure, *errno* is set to the following value:

[EFAULT] *arg* points outside the allocated address space.

I_FDINSERT creates a message from user specified buffer(s), adds information about another *stream* and sends the message downstream. The message contains a control part and an optional data part. The data and control

parts to be sent are distinguished by placement in separate buffers, as described below.

*arg* points to a *strfdinsert* structure which contains the following members:

```
struct strbuf   ctlbuf;
struct strbuf   databuf;
long            flags;
int             fd;
int             offset;
```

The *len* field in the *ctlbuf strbuf* structure [see *putmsg*(2)] must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. *fd* specifies the file descriptor of the other *stream* and *offset*, which must be word-aligned, specifies the number of bytes beyond the beginning of the control buffer where I_FDINSERT will store a pointer to the *fd stream*'s driver read queue structure. The *len* field in the *databuf strbuf* structure must be set to the number of bytes of data information to be sent with the message or zero if no data part is to be sent.

*flags* specifies the type of message to be created. A non-priority message is created if *flags* is set to 0, and a priority message is created if *flags* is set to RS_HIPRI. For non-priority messages, I_FDINSERT will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent. On failure, *errno* is set to one of the following values:

[EAGAIN] A non-priority message was specified, the O_NDELAY flag is set, and the *stream* write queue is full due to internal flow control conditions.

[EAGAIN] Buffers could not be allocated for the message that was to be created.

[EFAULT] *arg* points, or the buffer area specified in *ctlbuf* or *databuf* is, outside the allocated address space.

[EINVAL] One of the following: *fd* in the *strfdinsert* structure is not a valid, open *stream* file descriptor; the size of a pointer plus *offset* is greater than the *len* field for the buffer specified through *ctlptr*; *offset* does not specify a properly-aligned location in the data buffer; an undefined value is stored in *flags*.

[ENXIO] Hangup received on *fildes*.

[ERANGE] The *len* field for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module, or the *len* field for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message, or the *len* field for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

I_STR

Constructs an internal STREAMS ioctl message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send user *ioctl* requests to downstream modules and drivers. It allows information to be sent with the ioctl, and will return to the user any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with *errno* set to ETIME.

At most, one I_STR can be active on a *stream*. Further I_STR calls will block until the active I_STR completes at the *stream head*. The default timeout interval for these requests is 15 seconds. The O_NDELAY [see *open*(2)] flag has no effect on this call.

To send requests downstream, *arg* must point to a *strioctl* structure which contains the following members:

```
int     ic_cmd;         /* downstream command */
int     ic_timout;      /* ACK/NAK timeout */
int     ic_len;         /* length of data arg */
char    *ic_dp;         /* ptr to data arg */
```

*ic_cmd* is the internal ioctl command intended for a downstream module or driver and *ic_timout* is the number of seconds (-1 = infinite, 0 = use default, >0 = as specified) an I_STR request will wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument and *ic_dp* is a pointer to the data argument. The *ic_len* field has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the user (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the *stream* can return).

The *stream head* will convert the information pointed to by the *strioctl* structure to an internal ioctl command message and send it downstream. On failure, *errno* is set to one of the following values:

[EAGAIN] Unable to allocate buffers for the *ioctl* message.

[EFAULT] *arg* points, or the buffer area specified by *ic_dp* and *ic_len* (separately for data sent and data returned) is, outside the allocated address space.

[EINVAL] *ic_len* is less than 0 or *ic_len* is larger than the maximum configured size of the data part of a message or *ic_timout* is less than -1.

[ENXIO] Hangup received on *fildes*.

[ETIME] A downstream *ioctl* timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the *stream head*. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl command sent downstream fails. For these cases, I_STR will fail with *errno* set to the value in the message.

I_SENDFD

Requests the *stream* associated with *fildes* to send a message, containing a file pointer, to the *stream head* at the other end of a *stream* pipe. The

file pointer corresponds to *arg*, which must be an integer file descriptor.

I_SENDFD converts *arg* into the corresponding system file pointer. It allocates a message block and inserts the file pointer in the block. The user id and group id associated with the sending process are also inserted. This message is placed directly on the read queue [see *intro*(2)] of the *stream head* at the other end of the *stream* pipe to which it is connected. On failure, *errno* is set to one of the following values:

[EAGAIN] The sending *stream* is unable to allocate a message block to contain the file pointer.

[EAGAIN] The read queue of the receiving *stream head* is full and cannot accept the message sent by I_SENDFD.

[EBADF] *arg* is not a valid, open file descriptor.

[EINVAL] *fildes* is not connected to a *stream* pipe.

[ENXIO] Hangup received on *fildes*.

I_RECVFD Retrieves the file descriptor associated with the message sent by an I_SENDFD *ioctl* over a *stream* pipe. *arg* is a pointer to a data buffer large enough to hold an *strrecvfd* data structure containing the following members:

```
int fd;
unsigned short uid;
unsigned short gid;
char fill[8];
```

*fd* is an integer file descriptor. *uid* and *gid* are the user id and group id, respectively, of the sending *stream*.

If O_NDELAY is not set [see *open*(2)], I_RECVFD will block until a message is present at the *stream head*. If O_NDELAY is set, I_RECVFD will fail with *errno* set to EAGAIN if no message is present at the *stream head*.

If the message at the *stream head* is a message sent by an I_SENDFD, a new user file descriptor is allocated for the file pointer contained in the message. The new file descriptor is placed in the *fd* field of the *strrecvfd* structure. The structure is copied into the user data buffer pointed to by *arg*. On failure, *errno* is set to one of the following values:

[EAGAIN] A message was not present at the *stream head* read queue, and the O_NDELAY flag is set.

[EBADMSG] The message at the *stream head* read queue was not a message containing a passed file descriptor.

[EFAULT] *arg* points outside the allocated address space.

[EMFILE] NOFILES file descriptors are currently open.

[ENXIO] Hangup received on *fildes*.

The following two commands are used for connecting and disconnecting multiplexed STREAMS configurations.

I_LINK Connects two *streams*, where *fildes* is the file descriptor of the *stream* connected to the multiplexing driver, and *arg* is the file descriptor of the *stream* connected to another driver. The *stream* designated by *arg* gets

connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the *stream head* regarding the linking operation. This call returns a multiplexor ID number (an identifier used to disconnect the multiplexor, see I_UNLINK) on success, and a -1 on failure. On failure, *errno* is set to one of the following values:

[ENXIO] Hangup received on *fildes*.

[ETIME] Time out before acknowledgement message was received at *stream head*.

[EAGAIN] Unable to allocate STREAMS storage to perform the I_LINK.

[EBADF] *arg* is not a valid, open file descriptor.

[EINVAL] *fildes stream* does not support multiplexing.

[EINVAL] *arg* is not a *stream*, or is already linked under a multiplexor.

[EINVAL] The specified link operation would cause a "cycle" in the resulting configuration; that is, if a given *stream head* is linked into a multiplexing configuration in more than one place.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK will fail with *errno* set to the value in the message.

I_UNLINK Disconnects the two *streams* specified by *fildes* and *arg*. *fildes* is the file descriptor of the *stream* connected to the multiplexing driver. *arg* is the multiplexor ID number that was returned by the *ioctl* I_LINK command when a *stream* was linked below the multiplexing driver. If *arg* is -1, then all Streams which were linked to *fildes* are disconnected. As in I_LINK, this command requires the multiplexing driver to acknowledge the unlink. On failure, *errno* is set to one of the following values:

[ENXIO] Hangup received on *fildes*.

[ETIME] Time out before acknowledgement message was received at *stream head*.

[EAGAIN] Unable to allocate buffers for the acknowledgement message.

[EINVAL] Invalid multiplexor ID number.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the link request, if a message indicating an error or a hangup is received at the *stream head* of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK will fail with *errno* set to the value in the message.

**SEE ALSO**

close(2), fcntl(2), intro(2), ioctl(2), open(2), read(2), getmsg(2), poll(2), putmsg(2), signal(2), sigset(2), write(2) in the *Programmer's Reference Manual*.
*STREAMS Programmer's Guide*.
*STREAMS Primer*.

**DIAGNOSTICS**

Unless specified otherwise above, the return value from *ioctl* is 0 upon success and -1 upon failure with *errno* set as indicated.

**NAME**

sysconinit – system console initialization

**SYNOPSIS**

**/etc/sysconinit [-n]**

**DESCRIPTION**

The *sysconinit* command provides support for the linkage of */dev/console, /dev/syscon,* and */dev/systty* to the appropriate system console device. The choice of correct linkage is made from the *console* variable in the kernel and the existence of video hardware on workstations.

When the console variable is either 'T', 't', or '1', these devices are linked to /dev/tty1. When the variable is '0', these devices are linked to /dev/tty0. When the console variable is 'a', 'l', 'r', or 'g', these devices are linked to /dev/keybd when there is a video board in the system, and linked to /dev/tty0 without the video board.

The command is run a boot time from */etc/inittab,* before any access to the console has been made.

**FILES**

/dev/console
/dev/syscon
/dev/systty

**DIAGNOSTICS**

"sysconinit: [appropriate perror string]"
Printed when the uname system call fails.

"sysconinit: kopting console var: [appropriate perror string]"
Printed bsd43_syscall for BSD43_MIPS_KOPT fails.

"sysconinit: opening /dev/kmem: [appropriate perror string]"
Printed when open of /dev/kmem fails.

"sysconinit: seeking for 'console' value: [appropriate perror string]"
Printed when lseek to 'console' fails.

"sysconinit: reading for 'console' value: [appropriate perror string]"
Printed when read to 'console' fails.

"sysconinit: warning: '%c' unknown console setting, using 'l'"
Printed for bad values of console.

**NAME**

tcp – Internet Transmission Control Protocol

**SYNOPSIS**

None; included automatically with *inet*(4F).

**DESCRIPTION**

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. TCP provides for reliable interprocess communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit switched networks.

TCP fits into a layered protocol architecture just above the basic Internet Protocol (IP) described in *ip*(4P) which provides a way for TCP to send and receive variable-length segments of information enclosed in Internet datagram "envelopes." The Internet datagram provides a means for addressing source and destination TCPs in different networks, deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple netwokrs and interconnecting gateways, and has the ability to carry information on the precedence, security classification and compartmentalization of the TCP segments (although this is not currently implemented under the UNIX system.)

An application process interfaces to TCP through the *socket*(2) abstraction and the related calles *bind*(2), *listen*(2), *accept*(2), *connect*(2), *send*(2) and *recv*(2). The primary purpose of TCP is to provide a reliable bidirectional virtual circuit service between pairs of processes. In general, the TCP's decide when to block and forward data at their own convenience. In the UNIX system implementation, it is assumed that any buffering of data is done at the user level, and the TCP's transmit available data as soon as possible to their remote peer. They do this and always set the PUSH bit indicating that the transferred data should be made available to the user process at the remote end as soon as practicable.

To provide reliable data TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the underlying internet communications system. This is achieved by assigning a sequence number to each byte of data transmitted and requiring a positive acknowledgement from the receiving TCP. If the ACK is not received within an (adaptively determined) timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments. As long as the TCP's continue to function properly and the internet system does not become disjoint, no tranmission errors will affect the correct delivery of data, as TCP recovers from communications errors.

TCP provides flow control over the transmitted data. The receiving TCP is allowed to specify the amount of data which may be sent by the sender, by returning a *window* with every acknowledgement indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of bytes that the sender may transmit before receiving further permission.

TCP extends the standard 32-bit Internet host addresses with a 16-bit port number space; the combined addresses are available at the UNIX system process level in the standard *sockaddr_in* format described in *inet*(4F).

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*(2) system call must be used after binding the socket to an address with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) 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 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. See *inet*(4F) for a complete description of addressing in the Internet family.

A TCP connection is created at the server end by doing a *socket*(2), a *bind*(2) to establish the address of the socket, a *listen*(2) to cause connection queueing, and then an *accept*(2) which returns the descriptor for the socket. A client connects to the server by doing a *socket*(2) and then a *connect*(2). Data may then be sent from server to client and back using *read*(2V) and *write*(2V).

TCP implements a very weak out-of-band mechanism, which may be invoked using the out-of-band provisions of *send*(2). This mechanism allows setting an urgent pointer in the data stream; it is reflected to the TCP user by making the byte after the urgent pointer available as out-of-band data and providing a SIOCATMARK ioctl which returns an integer indicating whether the stream is at the urgent mark. The system never returns data across the urgent mark in a single read. Thus, when a SIGURG signal is received indicating the presence of out-of-band data, and the out-of-band data indicates that the data to the mark should be flushed (as in remote terminal processing), it suffices to loop, checking whether you are at the out-of-band mark, and reading data while you are not at the mark.

**SEE ALSO**

inet(4F), ip(4P)

**ERRORS**

It should be possible to send and receive TCP options.

The system always tries to negotiates the maximum TCP segment size to be 1024 bytes. This can result in poor performance if an intervening network performs excessive fragmentation.

SIOCSHIWAT and SIOCGHIWAT ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in <sys/ioctl.h>) but not implemented.

**NAME**

termio – general terminal interface

**DESCRIPTION**

All of the asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open terminal files; they are opened by *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, the buffer is flushed and all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character # erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character @ kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).

QUIT (Control-| or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

SWTCH (Control-z or ASCII SUB) is used by the job control facility, *shl*, to change the current layer to the control layer.

ERASE (#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF (Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (ASCII NUL) is an additional line delimiter, like NL. It is not normally used.

EOL2 is another additional line delimiter.

STOP (Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, SWTCH, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hang-up* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define NCC     8
struct  termio {
        unsigned  short   c_iflag;     /* input modes */
        unsigned  short   c_oflag;     /* output modes */
        unsigned  short   c_cflag;     /* control modes */
        unsigned  short   c_lflag;     /* local modes */
        char              c_line;      /* line discipline */
        unsigned  char    c_cc[NCC];   /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

```
0   VINTR   DEL
1   VQUIT   FS
2   VERASE  #
3   VKILL   @
4   VEOF    EOT
5   VEOL    NUL
6   reserved
```

7 SWTCH

The *c_iflag* field describes the basic terminal input control:

| | | |
|---|---|---|
| IGNBRK | 0000001 | Ignore break condition. |
| BRKINT | 0000002 | Signal interrupt on break. |
| IGNPAR | 0000004 | Ignore characters with parity errors. |
| PARMRK | 0000010 | Mark parity errors. |
| INPCK | 0000020 | Enable input parity check. |
| ISTRIP | 0000040 | Strip character. |
| INLCR | 0000100 | Map NL to CR on input. |
| IGNCR | 0000200 | Ignore CR. |
| ICRNL | 0000400 | Map CR to NL on input. |
| IUCLC | 0001000 | Map upper-case to lower-case on input. |
| IXON | 0002000 | Enable start/stop output control. |
| IXANY | 0004000 | Enable any character to restart output. |
| IXOFF | 0010000 | Enable start/stop input control. |

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character, will restart output which has been suspended.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all-bits-clear.

The *c_oflag* field specifies the system treatment of output:

| | | |
|---|---|---|
| OPOST | 0000001 | Postprocess output. |
| OLCUC | 0000002 | Map lower case to upper on output. |
| ONLCR | 0000004 | Map NL to CR-NL on output. |
| OCRNL | 0000010 | Map CR to NL on output. |
| ONOCR | 0000020 | No CR output at column 0. |
| ONLRET | 0000040 | NL performs CR function. |

| | | |
|---|---|---|
| OFILL | 0000100 | Use fill characters for delay. |
| OFDEL | 0000200 | Fill is DEL, else NUL. |
| NLDLY | 0000400 | Select new-line delays: |
| NL0 | 0 | |
| NL1 | 0000400 | |
| CRDLY | 0003000 | Select carriage-return delays: |
| CR0 | 0 | |
| CR1 | 0001000 | |
| CR2 | 0002000 | |
| CR3 | 0003000 | |
| TABDLY | 0014000 | Select horizontal-tab delays: |
| TAB0 | 0 | |
| TAB1 | 0004000 | |
| TAB2 | 0010000 | |
| TAB3 | 0014000 | Expand tabs to spaces. |
| BSDLY | 0020000 | Select backspace delays: |
| BS0 | 0 | |
| BS1 | 0020000 | |
| VTDLY | 0040000 | Select vertical-tab delays: |
| VT0 | 0 | |
| VT1 | 0040000 | |
| FFDLY | 0100000 | Select form-feed delays: |
| FF0 | 0 | |
| FF1 | 0100000 | |

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

If a form-feed or vertical-tab delay is specified, it lasts for about 2 seconds.

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

| | | |
|---|---|---|
| CBAUD | 0000017 | Baud rate: |
| B0 | 0 | Hang up |
| B50 | 0000001 | 50 baud |
| B75 | 0000002 | 75 baud |
| B110 | 0000003 | 110 baud |
| B134 | 0000004 | 134 baud |
| B150 | 0000005 | 150 baud |
| B200 | 0000006 | 200 baud |
| B300 | 0000007 | 300 baud |
| B600 | 0000010 | 600 baud |
| B1200 | 0000011 | 1200 baud |
| B1800 | 0000012 | 1800 baud |
| B2400 | 0000013 | 2400 baud |
| B4800 | 0000014 | 4800 baud |
| B9600 | 0000015 | 9600 baud |
| B19200 | 0000016 | 19200 baud |
| EXTA | 0000016 | External A |
| B38400 | 0000017 | 38400 baud |
| EXTB | 0000017 | External B |
| CSIZE | 0000060 | Character size: |
| CS5 | 0 | 5 bits |
| CS6 | 0000020 | 6 bits |
| CS7 | 0000040 | 7 bits |
| CS8 | 0000060 | 8 bits |
| CSTOPB | 0000100 | Send two stop bits, else one. |
| CREAD | 0000200 | Enable receiver. |
| PARENB | 0000400 | Parity enable. |
| PARODD | 0001000 | Odd parity, else even. |
| HUPCL | 0002000 | Hang up on last close. |
| CLOCAL | 0004000 | Local line, else dial-up. |
| RCV1EN | 0010000 | |
| XMT1EN | 0020000 | |
| LOBLK | 0040000 | Block layer output. |

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

If LOBLK is set, the output of a job control layer will be blocked when it is not the current layer. Otherwise the output generated by that layer will be multiplexed onto the current layer.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

| | | |
|---|---|---|
| ISIG | 0000001 | Enable signals. |
| ICANON | 0000002 | Canonical input (erase and kill processing). |
| XCASE | 0000004 | Canonical upper/lower presentation. |
| ECHO | 0000010 | Enable echo. |
| ECHOE | 0000020 | Echo erase character as BS-SP-BS. |
| ECHOK | 0000040 | Echo NL after kill character. |
| ECHONL | 0000100 | Echo NL. |
| NOFLSH | 0000200 | Disable flush after interrupt or quit. |

If ISIG is set, each input character is checked against the special control characters INTR, SWTCH, and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, read requests are satisfied directly from the input queue. A read will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired between characters. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

| *for*: | *use*: |
|---|---|
| ` | \' |
| \| | \! |
| ~ | \^ |
| { | \( |
| } | \) |
| \ | \\ |

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if

ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit, switch, and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.

TCSETA Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.

TCSETAW Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).

TCXONC Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.

TCFLSH If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

**FILES**

/dev/tty*

**SEE ALSO**

stty(1) in the *User's Reference Manual.*
fork(2), ioctl(2), setpgrp(2), signal(2) in the *Programmer's Reference Manual.*

**NAME**

tirdwr – Transport Interface read/write interface STREAMS module

**DESCRIPTION**

*tirdwr* is a STREAMS module that provides an alternate interface to a transport provider which supports the Transport Interface (TI) functions of the Network Services library (see Section 3N). This alternate interface allows a user to communicate with the transport protocol provider using the *read*(2) and *write*(2) system calls. The *putmsg*(2) and *getmsg*(2) system calls may also be used. However, *putmsg* and *getmsg* can only transfer data messages between user and *stream*.

The *tirdwr* module must only be pushed [see I_PUSH in *streamio*(7)] onto a *stream* terminated by a transport protocol provider which supports the TI. After the *tirdwr* module has been pushed onto a *stream*, none of the Transport Interface functions can be used. Subsequent calls to TI functions will cause an error on the *stream*. Once the error is detected, subsequent system calls on the *stream* will return an error with *errno* set to EPROTO.

The following are the actions taken by the *tirdwr* module when pushed on the *stream*, popped [see I_POP in *streamio*(7)] off the *stream*, or when data passes through it.

*push* – When the module is pushed onto a *stream*, it will check any existing data destined for the user to ensure that only regular data messages are present. It will ignore any messages on the *stream* that relate to process management, such as messages that generate signals to the user processes associated with the *stream*. If any other messages are present, the I_PUSH will return an error with *errno* set to EPROTO.

*write* – The module will take the following actions on data that originated from a *write* system call:

- All messages with the exception of messages that contain control portions (see the *putmsg* and *getmsg* system calls) will be transparently passed onto the module's downstream neighbor.
- Any zero length data messages will be freed by the module and they will not be passed onto the module's downstream neighbor.
- Any messages with control portions will generate an error, and any further system calls associated with the *stream* will fail with *errno* set to EPROTO.

*read* – The module will take the following actions on data that originated from the transport protocol provider:

- All messages with the exception of those that contain control portions (see the *putmsg* and *getmsg* system calls) will be transparently passed onto the module's upstream neighbor.
- The action taken on messages with control portions will be as follows:
  + Messages that represent expedited data will generate an error. All further system calls associated with the *stream* will fail with *errno* set to EPROTO.
  + Any data messages with control portions will have the control portions removed from the message prior to passing the message on to the upstream neighbor.
  + Messages that represent an orderly release indication from the transport provider will generate a zero length data message, indicating the end of file, which will be sent to the reader of the *stream*. The orderly release message itself will be freed by the module.
  + Messages that represent an abortive disconnect indication from the transport provider will cause all further *write* and *putmsg* system calls to fail

with *errno* set to ENXIO. All further *read* and *getmsg* system calls will return zero length data (indicating end of file) once all previous data has been read.

+ With the exception of the above rules, all other messages with control portions will generate an error and all further system calls associated with the *stream* will fail with *errno* set to EPROTO.

- Any zero length data messages will be freed by the module and they will not be passed onto the module's upstream neighbor.

*pop* – When popping the module off or closing the *stream*, the module will take the following actions:

- If an orderly release indication has been previously received, then an orderly release request will sent to the remote side of the transport connection.
- If an abortive disconnect has been previously received, then no action is taken.
- If neither an abortive disconnect nor an orderly release have been previously received, an abortive disconnect will be initiated by the module.
- If an error has occurred previously and an abortive disconnect has not been previously received, an abortive disconnect will be initiated by the module.

**SEE ALSO**

streamio(7), timod(7).
intro(2), getmsg(2), putmsg(2), read(2), write(2), intro(3) in the *Programmer's Reference Manual*.
*STREAMS Primer*.
*STREAMS Programmer's Guide*.
*Network Programmer's Guide*.

**NAME**

tty – controlling terminal interface

**DESCRIPTION**

The file */dev/tty* is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

**FILES**

/dev/tty
/dev/tty*

**SEE ALSO**

console(7), ports(7).

**NAME**

uart – RS2030/RC2030 IOP UART Driver

**SYNOPSIS**

**uart ports on RC2030 IOP**

**DESCRIPTION**

The *uart* driver supports the uarts on the R?2030 IOP. Except for the specialized *ioctl's* shown below, this driver is a standard serial stream terminal driver.

The following *ioctl* calls apply only to the */dev/tty[01]:*

| | |
|---|---|
| UTCSETTIMESTAMP | Turns on timestamping of all input from the uart. Each character received from the uart is presented in a timestamp structure. These are composed sync characters, the uart character, and the time in HZ since boot. The structure is defined in */usr/include/sys/uart_ioctl.h*. |
| UTCCLRTIMESTAMP | Turns off uart input timestamping. |

**FILES**

*/dev/tty[01]*

**SEE ALSO**

*tty(7)*

**NAME**

udp – Internet User Datagram Protocol

**SYNOPSIS**

None; comes automatically with *inet*(7F).

**DESCRIPTION**

The User Datagram Protocol (UDP) is defined to make available a datagram mode of packet switched computer communicaton in the environment of an interconnected set of computer networks. The protocol assumes that the Internet Protocol (IP) is used as the underlying protocol.

The protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP).

The UNIX system implementation of UDP makes it available as a socket of type SOCK_DGRAM. UDP sockets are normally used in a connectionless fashion, with the *sendto* and *recvfrom* calls described in *send*(2) and *recv*(2).

A UDP socket is created with a *socket*(2) call:

**s = socket(AF_INET, SOCK_DGRAM, 0);**

The socket initially has no address associated with it, and may be given an address with a *bind*(2) call as described in *inet*(7F). If no *bind* call is done, then the address assignment procedure described in *inet*(7F) is repeated as each datagram is sent.

When datagrams are sent the system encapsulates the user supplied data with UDP and IP headers. Unless the invoker is the super-user datagrams which would become broadcast packets on the network to which they are addressed are not allowed. Unless the socket has had a SO_DONTROUTE option enabled (see *socket*(2)) the outgoing datagram is routed through the routing tables as described in *routed*(1M). If there is insufficient system buffer space to temporarily hold the datagram while it is being trasmitted, the *sendto* may result in a ENOBUFS error. Other errors ( ENETUNREACH, EADDRNOTAVAIL, EACCES, EMSGSIZE ) may be generated by *icmp*(7P) or by the network interfaces themselves, and are reflected back in the *send* call.

As each UDP datagram arrives at a host the system strips out the IP options and checksums the data field, discarding the datagram if the checksum indicates that the datagram has been damaged. If no socket exists for the datagram to be sent to then an ICMP error is returned to the originating socket. If a socket exists for this datagram to be sent to, then we will append the datagram and the address from which it came to a queue associated with the datagram socket. This queue has limited capacity (2048 bytes of datagrams) and arriving datagrams which will not fit within its *high-water* capacity are silently discarded.

UDP processes ICMP errors reflected to it by *icmp*(7P). QUENCH errors are ignored (this is well considered a bug); UNREACH, TIMXCEED and PARAMPROB errors cause the socket to be disconnected from its peer if it was bound to a peer using *bind*(2) so that subsequent attempts to send datagrams via that socket will give an error indication.

The UDP datagram protocol differs from IP datagrams in that it adds a checksum over the data bytes and contains a 16-bit socket address on each machine rather than just the 32-bit machine address; UDP datagrams are addressed to sockets; IP packets are addressed to hosts.

**SEE ALSO**

recv(2), send(2), inet(7F)

"User Datagram Protocol," RFC768, John Postel, USC-ISI (Sun 800-1054-01)

**ERRORS**

SIOCSHIWAT and SIOCGHIWAT ioctl's to set and get the high water mark for the socket queue, and so that it can be changed from 2048 bytes to be larger or smaller, have been defined (in <sys/ioctl.h>) but not implemented.

Something sensible should be done with QUENCH errors if the socket is bound to a peer socket.

**NAME**

ac – login accounting

**SYNOPSIS**

**/etc/ac** [ **–w** wtmp ] [ **–p** ] [ **–d** ] [ people ] ...

**DESCRIPTION**

*ac* produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. **–w** is used to specify an alternate *wtmp* file. **–p** prints individual totals; without this option, only totals are printed. **–d** causes a printout for each midnight to midnight period. Any *people* will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/adm/wtmp* is used.

The accounting file */usr/adm/wtmp* is maintained by *init* and *login*. Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

**FILES**

/usr/adm/wtmp

**SEE ALSO**

init(8), sa(8), login(1), utmp(5).

**NAME**

bfsd – boot file system server

**SYNOPSIS**

**/etc/bfsd –d** *directory* [–u *username* ] [–g *groupname* ] [–i *interface* ]

**DESCRIPTION**

*bfsd* provides remote file access based on the SOCK_DGRAM networking interface to client systems. Typical uses include bringing in bootable images to a diskless network node. If *bfsd* is running on a machine with more than one network interface, it is able to gateway packets as required. This facility allows a node on one network to boot an image available on a machine physically connected to a second network.

The following command line arguments are supported.

**–d** *directory*
: Specifies the directory which the *bfsd* should consider its home. *bfsd* does a *chdir(2)* to that directory, and uses it as the root for all relative pathnames found in incoming requests.

**–u** *username*
: Attempt to run as the specified user. Typically this would be the user that owns the directory specified in the **–d** option.

**–g** *groupname*
: Attempt to run as a member of the specified group. Typically this would be the user that owns the directory specified in the **–d** option.

**–i** *interface*
: Specifies a directly connected network interface, such as *enp0*. If more than one interface is specified, then *bfsd* will forward packets among interfaces as necessary.

**SEE ALSO**

bfs(4P)

**BUGS**

*bfsd* should be started by the *inetd(8),* rather than operating independently. Two interfaces with different netmasks will break the packet forwarding machanism. The packet types related to the data write facility of the protocol are not implemented.

**NAME**

chown – change owner

**SYNOPSIS**

**/etc/chown** [ **–f –R** ] owner[.group] file ...

**DESCRIPTION**

*chown* changes the owner of the *files* to *owner*. The owner may be either a decimal UID or a login name found in the password file. An optional group may also be specified. The group may be either a decimal GID or a group name found in the group-ID file.

Only the super-user can change owner, in order to simplify accounting procedures. No errors are reported when the **–f** (force) option is given.

When the **–R** option is given, *chown* recursively descends its directory arguments setting the specified owner. When symbolic links are encountered, their ownership is changed, but they are not traversed.

**FILES**

/etc/passwd

**SEE ALSO**

chgrp(1), chown(2), passwd(5), group(5)

**NAME**

fingerd – remote user information server

**SYNOPSIS**

**/etc/fingerd**

**DESCRIPTION**

*fingerd* is a simple protocol based on RFC742 that provides an interface to the Name and Finger programs at several network sites. The program is supposed to return a friendly, human-oriented status report on either the system at the moment or a particular person in depth. There is no required format and the protocol consists mostly of specifying a single "command line".

*fingerd* listens for TCP requests at port 79. Once connected it reads a single command line terminated by a <CRLF> which is passed to *finger*(1). *fingerd* closes its connections as soon as the output is finished.

If the line is null (i.e. just a <CRLF> is sent) then *finger* returns a "default" report that lists all people logged into the system at that moment.

If a user name is specified (e.g. eric<CRLF>) then the response lists more extended information for only that particular user, whether logged in or not. Allowable "names" in the command line include both "login names" and "user names". If a name is ambiguous, all possible derivations are returned.

**SEE ALSO**

finger(1)

**BUGS**

Connecting directly to the server from a TIP or an equally narrow-minded TELNET-protocol user program can result in meaningless attempts at option negotiation being sent to the server, which will foul up the command line interpretation. *fingerd* should be taught to filter out IAC's and perhaps even respond negatively (IAC WON'T) to all option commands received.

**NAME**

lockd – network lock daemon

**SYNOPSIS**

**/etc/rpc.lockd** [ **–t** *timeout* ] [ **–g** *graceperiod* ]

**DESCRIPTION**

*lockd* processes lock requests that are either sent locally by the kernel or remotely by another lock daemon. *lockd* forwards lock requests for remote data to the server site's lock daemon through the RPC/XDR(3N) package. *lockd* then requests the status monitor daemon, *statd(8C),* for monitor service. The reply to the lock request will not be sent to the kernel until the status daemon and the server site's lock daemon have replied.

If either the status monitor or server site's lock daemon is unavailable, the reply to a lock request for remote data is delayed until all daemons become available.

When a server recovers, it waits for a grace period for all client site lockds to submit reclaim requests. Client site lockds, on the other hand, are notified by the statd of the server recovery and promptly resubmit previously granted lock requests. If a lockd fails to secure a previously granted lock at the server site, the *lockd* sends SIGLOST to a process.

**OPTIONS**

**–t** *timeout*

*lockd* uses *timeout* (seconds) as the interval instead of the default value (15 seconds) to retransmit lock request to the remote server.

**–g** *graceperiod*

*lockd* uses *graceperiod* (seconds) as the grace period duration instead of the default value (45 seconds).

**SEE ALSO**

fcntl(2), lockf(3), signal(3), statd(8C) a.

**NAME**

lpc – line printer control program

**SYNOPSIS**

**/etc/lpc** [ command [ argument ... ] ]

**DESCRIPTION**

*lpc* is used by the system administrator to control the operation of the line printer system. For each line printer configured in /etc/printcap, *lpc* may be used to:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer dameons.

Without any arguments, *lpc* will prompt for commands from the standard input. If arguments are supplied, *lpc* interprets the first argument as a command and the remaining arguments as parameters to the command. The standard input may be redirected causing *lpc* to read commands from file. Commands may be abreviated; the following is the list of recognized commands.

? [ command ... ]

help [ command ... ]
: Print a short description of each command specified in the argument list, or, if no arguments are given, a list of the recognized commands.

abort { all | printer ... }
: Terminate an active spooling daemon on the local host immediately and then disable printing (preventing new daemons from being started by *lpr*) for the specified printers.

clean { all | printer ... }
: Remove any temporary files, data files, and control files that cannot be printed (i.e., do not form a complete printer job) from the specified printer queue(s) on the local machine.

disable { all | printer ... }
: Turn the specified printer queues off. This prevents new printer jobs from being entered into the queue by *lpr*.

down { all | printer } message ...
: Turn the specified printer queue off, disable printing and put *message* in the printer status file. The message doesn't need to be quoted, the remaining arguments are treated like *echo*(1). This is normally used to take a printer down and let others know why (lpq will indicate the printer is down and print the status message).

enable { all | printer ... }
: Enable spooling on the local queue for the listed printers. This will allow *lpr* to put new jobs in the spool queue.

exit

quit
: Exit from lpc.

restart { all | printer ... }
: Attempt to start a new printer daemon. This is useful when some abnormal condition causes the daemon to die unexpectedly leaving jobs in the queue. *lpq* will report that there is no daemon present when this condition occurs. If the user is the super-user, try to abort the current daemon first (i.e., kill and restart a stuck daemon).

start { all | printer ... }
Enable printing and start a spooling daemon for the listed printers.

status { all | printer ... }
Display the status of daemons and queues on the local machine.

stop { all | printer ... }
Stop a spooling daemon after the current job completes and disable printing.

topq printer [ jobnum ... ] [ user ... ]
Place the jobs in the order listed at the top of the printer queue.

up { all | printer ... }
Enable everything and start a new printer daemon. Undoes the effects of *down*.

**FILES**

| | |
|---|---|
| /etc/printcap | printer description file |
| /usr/spool/* | spool directories |
| /usr/spool/*/lock | lock file for queue control |

**SEE ALSO**

lpd(8), lpr(1), lpq(1), lprm(1), printcap(5)

**DIAGNOSTICS**

| | |
|---|---|
| ?Ambiguous command | abreviation matches more than one command |
| ?Invalid command | no match was found |
| ?Privileged command | command can be executed by root only |

**NAME**

lpd – line printer daemon

**SYNOPSIS**

**/usr/lib/lpd** [ **–l** ] [ port # ]

**DESCRIPTION**

*lpd* is the line printer daemon (spool area handler) and is normally invoked at boot time from the *rc*(8) file. It makes a single pass through the *printcap*(5) file to find out about the existing printers and prints any files left after a crash. It then uses the system calls *listen*(2) and *accept*(2) to receive requests to print files in the queue, transfer files to the spooling area, display the queue, or remove jobs from the queue. In each case, it forks a child to handle the request so the parent can continue to listen for more requests. The Internet port number used to rendezvous with other processes is normally obtained with *getservbyname*(3) but can be changed with the *port#* argument. The **–l** flag causes *lpd* to log valid requests received from the network. This can be useful for debugging purposes.

Access control is provided by two means. First, All requests must come from one of the machines listed in the file */etc/hosts.equiv* or */etc/hosts.lpd*. Second, if the "rs" capability is specified in the *printcap* entry for the printer being accessed, *lpr* requests will only be honored for those users with accounts on the machine with the printer.

The file *minfree* in each spool directory contains the number of disk blocks to leave free so that the line printer queue won't completely fill the disk. The *minfree* file can be edited with your favorite text editor.

The file *lock* in each spool directory is used to prevent multiple daemons from becoming active simultaneously, and to store information about the daemon process for *lpr*(1), *lpq*(1), and *lprm*(1). After the daemon has successfully set the lock, it scans the directory for files beginning with *cf*. Lines in each *cf* file specify files to be printed or non-printing actions to be performed. Each such line begins with a key character to specify what to do with the remainder of the line.

J  Job Name. String to be used for the job name on the burst page.

C  Classification. String to be used for the classification line on the burst page.

L  Literal. The line contains identification info from the password file and causes the banner page to be printed.

T  Title. String to be used as the title for *pr*(1).

H  Host Name. Name of the machine where *lpr* was invoked.

P  Person. Login name of the person who invoked *lpr*. This is used to verify ownership by *lprm*.

M  Send mail to the specified user when the current print job completes.

f  Formatted File. Name of a file to print which is already formatted.

l  Like "f" but passes control characters and does not make page breaks.

p  Name of a file to print using *pr*(1) as a filter.

t  Troff File. The file contains *troff*(1) output (cat phototypesetter commands).

n  Ditroff File. The file contains device independent troff output.

d  DVI File. The file contains *Tex*(l) output (DVI format from Standford).

g  Graph File. The file contains data produced by *plot*(3X).

c  Cifplot File. The file contains data produced by *cifplot*.

v  The file contains a raster image.

r The file contains text data with FORTRAN carriage control characters.

1 Troff Font R. Name of the font file to use instead of the default.

2 Troff Font I. Name of the font file to use instead of the default.

3 Troff Font B. Name of the font file to use instead of the default.

4 Troff Font S. Name of the font file to use instead of the default.

W Width. Changes the page width (in characters) used by *pr*(1) and the text filters.

I Indent. The number of characters to indent the output by (in ascii).

U Unlink. Name of file to remove upon completion of printing.

N File name. The name of the file which is being printed, or a blank for the standard input (when *lpr* is invoked in a pipeline).

If a file can not be opened, a message will be logged via *syslog*(3) using the *LOG_LPR* facility. *lpd* will try up to 20 times to reopen a file it expects to be there, after which it will skip the file to be printed.

*lpd* uses *flock*(2) to provide exclusive access to the lock file and to prevent multiple deamons from becoming active simultaneously. If the daemon should be killed or die unexpectedly, the lock file need not be removed. The lock file is kept in a readable ASCII form and contains two lines. The first is the process id of the daemon and the second is the control file name of the current job being printed. The second line is updated to reflect the current status of *lpd* for the programs *lpq*(1) and *lprm*(1).

**FILES**

| | |
|---|---|
| /etc/printcap | printer description file |
| /usr/spool/* | spool directories |
| /usr/spool/*/minfree | minimum free space to leave |
| /dev/lp* | line printer devices |
| /dev/printer | socket for local requests |
| /etc/hosts.equiv | lists machine names allowed printer access |
| /etc/hosts.lpd | lists machine names allowed printer access, but not under same administrative control. |

**SEE ALSO**

lpc(8), pac(1), lpr(1), lpq(1), lprm(1), syslog(3), printcap(5)
*4.2BSD Line Printer Spooler Manual*

**NAME**

named – Internet domain name server

**SYNOPSIS**

**named** [ **–d** *debuglevel* ] [ **–p** *port#* ] [ *bootfile* ]

**DESCRIPTION**

*named* is the Internet domain name server (see RFC883 for more details). Without any arguments, *named* will read the default boot file */etc/named.boot*, read any initial data and listen for queries.

Options are:

**–d** Print debugging information. A number after the "d" determines the level of messages printed.

**–p** Use a different port number. The default is the standard port number as listed in /etc/services.

Any additional argument is taken as the name of the boot file. The boot file contains information about where the name server is to get its initial data. The following is a small example:

```
;
;       boot file for name server
;
; type          domain          source file or host
;
domain          berkeley.edu
primary         berkeley.edu    named.db
secondary       cc.berkeley.edu 10.2.0.78 128.32.0.10
cache                           named.ca
```

The first line specifies that "berkeley.edu" is the domain for which the server is authoritative. The second line states that the file "named.db" contains authoritative data for the domain "berkeley.edu". The file "named.db" contains data in the master file format described in RFC883 except that all domain names are relative to the origin; in this case, "berkeley.edu" (see below for a more detailed description). The second line specifies that all authoritative data under "cc.berkeley.edu" is to be transferred from the name server at 10.2.0.78. If the transfer fails it will try 128.32.0.10 and continue trying the address, up to 10, listed on this line. The secondary copy is also authoritative for the specified domain. The fourth line specifies data in "named.ca" is to be placed in the cache (i.e., well known data such as locations of root domain servers). The file "named.ca" is in the same format as "named.db".

The master file consists of entries of the form:

```
$INCLUDE <filename>
$ORIGIN <domain>
<domain> <opt_ttl> <opt_class> <type> <resource_record_data>
```

where *domain* is "." for root, "@" for the current origin, or a standard domain name. If *domain* is a standard domain name that does not end with ".", the current origin is appended to the domain. Domain names ending with "." are unmodified. The *opt_ttl* field is an optional integer number for the time-to-live field. It defaults to zero. The *opt_class* field is the object address type; currently only one type is supported, **IN**, for objects connected to the DARPA Internet. The *type* field is one of the following tokens; the data expected in the *resource_record_data* field is in parentheses.

| | |
|---|---|
| A | a host address (dotted quad) |
| NS | an authoritative name server (domain) |
| MX | a mail exchanger (domain) |
| CNAME | the canonical name for an alias (domain) |
| SOA | marks the start of a zone of authority (5 numbers (see RFC883)) |
| MB | a mailbox domain name (domain) |
| MG | a mail group member (domain) |
| MR | a mail rename domain name (domain) |
| NULL | a null resource record (no format or data) |
| WKS | a well know service description (not implemented yet) |
| PTR | a domain name pointer (domain) |
| HINFO | host information (cpu_type OS_type) |
| MINFO | mailbox or mail list information (request_domain error_domain) |

**NOTES**

The following signals have the specified effect when sent to the server process using the *kill*(1) command.

| | |
|---|---|
| SIGHUP | Causes server to read named.boot and reload database. |
| SIGINT | Dumps current data base and cache to /usr/tmp/named_dump.db |
| SIGUSR1 | Turns on debugging; each SIGUSR1 increments debug level. |
| SIGUSR2 | Turns off debugging completely. |

**FILES**

| | |
|---|---|
| /etc/named.boot | name server configuration boot file |
| /etc/named.pid | the process id |
| /usr/tmp/named.run | debug output |
| /usr/tmp/named_dump.db | dump of the name servers database |

**SEE ALSO**

kill(1), gethostbyname(3N), signal(3c), resolver(3), resolver(5), RFC882, RFC883, RFC973, RFC974, *Name Server Operations Guide for BIND*

**NAME**

sprayd – spray server

**SYNOPSIS**

**/usr/etc/rpc.sprayd**

**DESCRIPTION**

*rpc.sprayd* is a server which records the packets sent by *spray*(8). The *rpc.sprayd* daemon is normally invoked by *inetd*(8C).

**SEE ALSO**

spray(8)

**NAME**

syslogd – log systems messages

**SYNOPSIS**

**/etc/syslogd** [ **–f***configfile* ] [ **–m***markinterval* ] [ **–d** ]

**DESCRIPTION**

*syslogd* reads and logs messages into a set of files described by the configuration file /etc/syslog.conf. Each message is one line. A message can contain a priority code, marked by a number in angle braces at the beginning of the line. Priorities are defined in *<sys/syslog.h>*. *syslogd* reads from the UNIX domain socket */dev/log*, from an Internet domain socket specified in */etc/services*, and from the special device */dev/klog* (to read kernel messages).

*syslogd* configures when it starts up and whenever it receives a hangup signal. Lines in the configuration file have a *selector* to determine the message priorities to which the line applies and an *action*. The *action* field are separated from the selector by one or more tabs.

Selectors are semicolon separated lists of priority specifiers. Each priority has a *facility* describing the part of the system that generated the message, a dot, and a *level* indicating the severity of the message. Symbolic names may be used. An asterisk selects all facilities. All messages of the specified level or higher (greater severity) are selected. More than one facility may be selected using commas to separate them. For example:

*.emerg;mail,daemon.crit

Selects all facilities at the *emerg* level and the *mail* and *daemon* facilities at the *crit* level.

Known facilities and levels recognized by *syslogd* are those listed in *syslog*(3) without the leading "LOG_". The additional facility "mark" has a message at priority LOG_INFO sent to it every 20 minutes (this may be changed with the **–m** flag). The "mark" facility is not enabled by a facility field containing an asterisk. The level "none" may be used to disable a particular facility. For example,

*.debug;mail.none

Sends all messages *except* mail messages to the selected file.

The second part of each line describes where the message is to be logged if this line is selected. There are four forms:

- A filename (beginning with a leading slash). The file will be opened in append mode. If the filename is preceded by a ?, as in the name "?/dev/console", messages will only be logged to that file if there is no user logged on to that device. This only applies to files in the directory /dev.
- A hostname preceded by an at sign ("@"). Selected messages are forwarded to the *syslogd* on the named host.
- A comma separated list of users. Selected messages are written to those users if they are logged in.
- An asterisk. Selected messages are written to all logged-in users.

Blank lines and lines beginning with '#' are ignored.

For example, the configuration file:

```
kern,mark.debug         /dev/console
*.notice;mail.info      /usr/spool/adm/syslog
*.crit                  /usr/adm/critical
kern.err                @ucbarpa
*.emerg                 *
```

```
*.alert                 eric,kridle
*.alert;auth.warning    ralph
```

logs all kernel messages and 20 minute marks onto the system console, all notice (or higher) level messages and all mail system messages except debug messages into the file /usr/spool/adm/syslog, and all critical messages into /usr/adm/critical; kernel messages of error severity or higher are forwarded to ucbarpa. All users will be informed of any emergency messages, the users "eric" and "kridle" will be informed of any alert messages, and the user "ralph" will be informed of any alert message, or any warning message (or higher) from the authorization system.

In cases where the console is a CRT terminal, it may not be desirable to log messages to the console when a user is logged in. This problem can be solved by setting up entries like the following:

```
kern,mark.debug         ?/dev/console
kern,mark.debug         /usr/adm/console_log
```

This tells the daemon to log kernel messages and marks to /dev/console unless someone is logged in on the console, and to log the same messages to the file /usr/adm/console_log unconditionally.

The flags are:

**-f** Specify an alternate configuration file.

**-m** Select the number of minutes between mark messages.

**-d** Turn on debugging.

*syslogd* creates the file /etc/syslog.pid, if possible, containing a single line with its process id. This can be used to kill or reconfigure *syslogd*.

To bring *syslogd* down, it should be sent a terminate signal (e.g. kill `cat /etc/syslog.pid`).

**FILES**

| | |
|---|---|
| /etc/syslog.conf | the configuration file |
| /etc/syslog.pid | the process id |
| /dev/log | Name of the UNIX domain datagram log socket |
| /dev/klog | The kernel log device |

**SEE ALSO**

logger(1), syslog(3)

**NAME**

talkd – remote user communication server

**SYNOPSIS**

**/etc/talkd**

**DESCRIPTION**

*talkd* is the server that notifies a user that somebody else wants to initiate a conversation. It acts a repository of invitations, responding to requests by clients wishing to rendezvous to hold a conversation. In normal operation, a client, the caller, initiates a rendezvous by sending a CTL_MSG to the server of type LOOK_UP (see *<protocols/talkd.h>*). This causes the server to search its invitation tables to check if an invitation currently exists for the caller (to speak to the callee specified in the message). If the lookup fails, the caller then sends an ANNOUNCE message causing the server to broadcast an announcement on the callee's login ports requesting contact. When the callee responds, the local server uses the recorded invitation to respond with the appropriate rendezvous address and the caller and callee client programs establish a stream connection through which the conversation takes place.

**SEE ALSO**

talk(1), write(1)

**NAME**

timed – time server daemon

**SYNOPSIS**

**/etc/timed** [ -t ] [ -M ] [ -n network ] [ -i network ]

**DESCRIPTION**

*timed* is the time server daemon and is normally invoked at boot time from the *rc*(8) file. It synchronizes the host's time with the time of other machines in a local area network running *timed*(8). These time servers will slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time. The average network time is computed from measurements of clock differences using the ICMP timestamp request message.

The service provided by *timed* is based on a master-slave scheme. When *timed*(8) is started on a machine, it asks the master for the network time and sets the host's clock to that time. After that, it accepts synchronization messages periodically sent by the master and calls *adjtime*(2) to perform the needed corrections on the host's clock.

It also communicates with *date*(1) in order to set the date globally, and with *timedc*(8), a timed control program. If the machine running the master crashes, then the slaves will elect a new master from among slaves running with the **–M** flag. A *timed* running without the **–M** flag will remain a slave. The **–t** flag enables *timed* to trace the messages it receives in the file /usr/adm/timed.log. Tracing can be turned on or off by the program *timedc*(8). *timed* normally checks for a master time server on each network to which it is connected, except as modified by the options described below. It will request synchronization service from the first master server located. If permitted by the **–M** flag, it will provide synchronization service on any attached networks on which no current master server was detected. Such a server propagates the time computed by the top-level master. The **–n** flag, followed by the name of a network which the host is connected to (see *networks*(5)), overrides the default choice of the network addresses made by the program. Each time the **–n** flag appears, that network name is added to a list of valid networks. All other networks are ignored. The **–i** flag, followed by the name of a network to which the host is connected (see *networks*(5)), overrides the default choice of the network addresses made by the program. Each time the **–i** flag appears, that network name is added to a list of networks to ignore. All other networks are used by the time daemon. The **–n** and **–i** flags are meaningless if used together.

**FILES**

| | |
|---|---|
| /usr/adm/timed.log | tracing file for timed |
| /usr/adm/timed.masterlog | log file for master timed |

**SEE ALSO**

date(1), adjtime(2), gettimeofday(2), icmp(4P), timedc(8),
*TSP: The Time Synchronization Protocol for UNIX 4.3BSD*, R. Gusella and S. Zatti

**NAME**

vipw – edit the password file

**SYNOPSIS**

**vipw**

**DESCRIPTION**

*vipw* edits the password file while setting the appropriate locks, and does any necessary processing after the password file is unlocked. If the password file is already being edited, then you will be told to try again later. The *vi* editor will be used unless the environment variable EDITOR indicates an alternate editor. *vipw* performs a number of consistency checks on the password entry for *root*, and will not allow a password file with a "mangled" root entry to be installed.

**SEE ALSO**

passwd(1), passwd(5), adduser(8), mkpasswd(8)

**FILES**

/etc/ptmp

**NAME**

zdump – time zone dumper

**SYNOPSIS**

**zdump** [ **–v** ] [ **–c** cutoffyear ] [ zonename ... ]

**DESCRIPTION**

*zdump* prints the current time in each *zonename* named on the command line.

These options are available:

**–v** For each *zonename* on the command line, print the current time, the time at the lowest possible time value, the time one day after the lowest possible time value, the times both one second before and exactly at each time at which the rules for computing local time change, the time at the highest possible time value, and the time at one day less than the highest possible time value. Each line ends with **isdst=1** if the given time is Daylight Saving Time or **isdst=0** otherwise.

**–c** *cutoffyear*
Cut off the verbose output near the start of the given year.

**FILES**

/etc/zoneinfo standard zone information directory

**SEE ALSO**

date(1), ctime(3), tzfile(5), zic(8).

**NAME**

zic – time zone compiler

**SYNOPSIS**

**zic** [ **–v** ] [ **–d** *directory* ] [ **–l** *localtime* ] [ *filename ...* ]

**DESCRIPTION**

*zic* reads text from the file(s) named on the command line and creates the time conversion information files specified in this input. If a *filename* is **–**, the standard input is read.

*zic* can also be used to set the timezone. When your machine is installed or updated, the timezone information found in /etc/zoneinfo/SOURCES is automatically compiled for you. If the file /etc/zoneinfo/localtime does not exist, the timezone is set to US/Pacific time. To change the timezone, look in the directory /etc/zoneinfo, determine your timezone (you can use an abbreviation like PST, or a long name, just as long as it is the name of a file found in /etc/zoneinfo), and execute the command "/etc/zic -l timezone", where "timezone" is the name of the timezone you need. The timezone need only be set once.

These options are available:

**–d** *directory*
Create time conversion information files in the named directory rather than in the standard directory named below.

**–l** *timezone*
Use the given time zone as local time. *zic* will act as if the file contained a link line of the form

Link *timezone* localtime

**–v** Complain if a year that appears in a data file is outside the range of years representable by *time*(2) values.

Input lines are made up of fields. Fields are separated from one another by any number of white space characters. Leading and trailing white space on input lines is ignored. An unquoted sharp character (#) in the input introduces a comment which extends to the end of the line the sharp character appears on. White space characters and sharp characters may be enclosed in double quotes (") if they're to be used as part of a field. Any line that is blank (after comment stripping) is ignored. Non-blank lines are expected to be of one of three types: rule lines, zone lines, and link lines.

A rule line has the form

**Rule NAME FROM TO TYPE IN ON AT SAVE LETTER/S**

For example:

Rule USA 1969 1973 – Apr lastSun 2:00 1:00 D

The fields that make up a rule line are:

**NAME** Gives the (arbitrary) name of the set of rules this rule is part of.

**FROM** Gives the first year in which the rule applies. The word **minimum** (or an abbreviation) means the minimum year with a representable time value. The word **maximum** (or an abbreviation) means the maximum year with a representable time value.

**TO** Gives the final year in which the rule applies. In addition to **minimum** and

**maximum** (as above), the word **only** (or an abbreviation) may be used to repeat the value of the **FROM** field.

**TYPE** Gives the type of year in which the year applies. If **TYPE** is **–** then the rule applies in all years between **FROM** and **TO** inclusive; if **TYPE** is **uspres**, the rule applies in U.S. Presidential election years; if **TYPE** is **nonpres**, the rule applies in years other than U.S. Presidential election years. If **TYPE** is something else, then *zic* executes the command

**yearistype** *year type*

to check the type of a year: an exit status of zero is taken to mean that the year is of the given type; an exit status of one is taken to mean that the year is not of the given type.

**IN** Names the month in which the rule takes effect. Month names may be abbreviated.

**ON** Gives the day on which the rule takes effect. Recognized forms include:

| | |
|---|---|
| 5 | the fifth of the month |
| lastSun | the last Sunday in the month |
| lastMon | the last Monday in the month |
| Sun>=8 | first Sunday on or after the eighth |
| Sun<=25 | last Sunday on or before the 25th |

Names of days of the week may be abbreviated or spelled out in full. Note that there must be no spaces within the **ON** field.

**AT** Gives the time of day at which the rule takes affect. Recognized forms include:

| | |
|---|---|
| 2 | time in hours |
| 2:00 | time in hours and minutes |
| 15:00 | 24-hour format time (for times after noon) |
| 1:28:14 | time in hours, minutes, and seconds |

Any of these forms may be followed by the letter **w** if the given time is local "wall clock" time or **s** if the given time is local "standard" time; in the absence of **w** or **s**, wall clock time is assumed.

**SAVE** Gives the amount of time to be added to local standard time when the rule is in effect. This field has the same format as the **AT** field (although, of course, the **w** and **s** suffixes are not used).

**LETTER/S**

Gives the "variable part" (for example, the "S" or "D" in "EST" or "EDT") of time zone abbreviations to be used when this rule is in effect. If this field is **–**, the variable part is null.

A zone line has the form

Zone NAME GMTOFF RULES/SAVE FORMAT [UNTIL]

For example:

| | | | | | |
|---|---|---|---|---|---|
| Zone | Australia/South-west | 9:30 | Aus | CST | 1987 Mar 15 2:00 |

The fields that make up a zone line are:

**NAME** The name of the time zone. This is the name used in creating the time conversion information file for the zone.

**GMTOFF**

The amount of time to add to GMT to get standard time in this zone. This field has the same format as the **AT** and **SAVE** fields of rule lines; begin the field with a minus sign if time must be subtracted from GMT.

**RULES/SAVE**

The name of the rule(s) that apply in the time zone or, alternately, an amount of time to add to local standard time. If this field is **–** then standard time always applies in the time zone.

**FORMAT**

The format for time zone abbreviations in this time zone. The pair of characters **%s** is used to show where the "variable part" of the time zone abbreviation goes. **UNTIL** The time at which the GMT offset or the rule(s) change for a location. It is specified as a year, a month, a day, and a time of day. If this is specified, the time zone information is generated from the given GMT offset and rule change until the time specified.

The next line must be a "continuation" line; this has the same form as a zone line except that the string "Zone" and the name are omitted, as the continuation line will place information starting at the time specified as the **UNTIL** field in the previous line in the file used by the previous line. Continuation lines may contain an **UNTIL** field, just as zone lines do, indicating that the next line is a further continuation.

A link line has the form

```
Link  LINK-FROM  LINK-TO
```

For example:

```
Link  US/Eastern    EST5EDT
```

The **LINK-FROM** field should appear as the **NAME** field in some zone line; the **LINK-TO** field is used as an alternate name for that zone.

Except for continuation lines, lines may appear in any order in the input.

**NOTE**

For areas with more than two types of local time, you may need to use local standard time in the **AT** field of the earliest transition time's rule to ensure that the earliest transition time recorded in the compiled file is correct.

**FILES**

/etc/zoneinfo  standard directory used for created files

**SEE ALSO**

date(1), ctime(3), tzfile(5), zdump(8).