IRIX Admin: Selected Reference Pages

Document Number 007-2159-005

CONTRIBUTORS

Edited by Susan Ellis
Production by Lorrie Williams
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

IRIX Admin: Selected Reference Pages
Document Number 007-2159-005

About This Guide

This volume contains selected reference pages on IRIX system administration topics. The reference pages included are those that may be useful when a system is down and online reference pages are unavailable.

The reference pages are organized by section and appear alphabetically within each section. Pages that discuss more than one topic are sorted by the first topic on the page. The tables below list each reference page:

- Table i lists section 1 and 1M reference pages for commands and system maintenance utilities.
- Table ii lists section 4 reference pages for file formats.
- Table iii lists one section 5 reference page for a special facility, availmon.
- Table iv lists section 7 and 7M reference pages for protocols and special files.

Table i Commands and System Maintenance

Reference Page	Function	Page
Add_disk(1)	add an optional disk to the system	1
amreport(1M)	produce statistical and availability reports from availmon log file	2
autoconfig(1M)	configure kernel	4
Backup(1)	backup the specified file or directory	6
bcheckrc(1M)	system initialization procedures	12
bootp(1M)	server for Internet Bootstrap Protocol	7
brc(1M)	system initialization procedures	12
chkconfig(1M)	configuration state checker	13
chroot(1M)	change root directory for a command	17

Table i (continued) Commands and System Maintenance

Reference Page	Function	Page
csh(1)	shell command interpreter with a C-like syntax	18
devnm(1M)	device name	44
dvhtool(1M)	modify and obtain disk volume header information	45
ecc(1)	dump memory ECC log	47
ed(1)	text editor	48
find(1)	find files	59
fsck(1M)	check and repair filesystems for EFS	62
fsdb(1M)	filesystem debugger for EFS	66
fsstat(1M)	report filesystem status	75
ftp(1C)	Internet file transfer program	76
fx(1M)	disk utility	87
growfs(1M)	expand a filesystem	102
hinv(1M)	hardware inventory command	104
icrash(1M)	IRIX system crash analysis utility	105
ifconfig(1M)	configure network interface parameters	113
init(1M)	process control initialization	115
inst(1M)	software installation tool	120
jsh(1)	shell, the standard/job control/restricted command programming language	243
killall(1M)	kill named processes	155
lboot(1M)	configure bootable kernel	157
login(1)	sign on	160
lvck (1M)	check and restore consistency of logical volumes	167
lvinfo(1M)	print information about active logical volumes	171

Table i (continued) Commands and System Maintenance

Reference Page	Function	Page
lvinit(1M)	initialize logical volume devices	173
MAKEDEV(1M)	create device special files	174
mkfs(1M)	construct a filesystem	177
mkfs_efs(1M)	construct an EFS filesystem	178
mkfs_xfs(1M)	construct an XFS filesystem	182
mklv(1M)	construct or extend a logical volume	187
ml(1M)	load dynamic kernel modules	190
mount(1M)	mount and unmount filesystems	193
network(1M)	network initialization and shutdown script	198
nvram(1M)	get or set non-volatile RAM variables	202
prom(1M)	PROM monitor	204
prtvtoc(1M)	print disk volume header information	214
ps(1)	report process status	217
pwck(1M)	password file checker	223
pwconv(1M)	install and update /etc/shadow with information from /etc/passwd	225
rcp(1C)	remote file copy	227
red(1)	text editor	48
Restore(1)	restore the specified file or directory from tape	229
restore(1M)	incremental filesystem restore	230
rlogin(1C)	remote login	236
rsh(1)	shell, the standard/job control/restricted command programming language	243
rsh(1C)	remote shell	238
rrestore(1M)	incremental filesystem restore	230

Table i (continued) Commands and System Maintenance

Reference Page	Function	Page
savecore(1M)	save a crash vmcore dump of the operating system	240
setmnt(1M)	establish mount table	242
sgikopt(1M)	get or set non-volatile RAM variables	202
sh(1)	shell, the standard/job control/restricted command programming language	243
statd(1M)	network status monitor daemon	260
su(1M)	become superuser or another user	261
symmon(1M)	kernel symbolic debugger	264
systune(1M)	display and set tunable parameters	269
telinit(1M)	process control initialization	115
telnet(1C)	user interface to the TELNET protocol	272
tftpd(1M)	internet Trivial File Transfer Protocol server	281
umount(1M)	mount and unmount filesystems	193
versions(1M)	software versions tool	283
xfsrestore(1M)	XFS filesystem incremental restore utility	289
xfs_check(1M)	check XFS filesystem consistency	296
xfs_growfs(1M)	expand an XFS filesystem	299
xlv_make(1M)	create logical volume objects	301
xlv_mgr(1M)		312

Table ii File Formats

Reference Page	Function	Page
core(4)	format of core image file	
efs(4)	layout of the Extent File System	
fstab(4)	static information about filesystems	

Table ii (continued) File Formats

Reference Page	Function	Page
gettydefs(4)	speed and terminal settings used by getty	
hosts(4)	hostname-address database	
inittab(4)	script for the init process	
inode(4)	format of an Extent File System inode	
lvtab(4)	information about logical volumes	
master(4)	master configuration database	
mload(4)	dynamically loadable kernel modules	
mtune(4)	default system tunable parameters	
passwd(4)	password file	
profile(4)	setting up an environment at login time	
shadow(4)	shadow password file	
stune(4)	local settings for system tunable parameters	
system(4)	system configuration information directory	
sys_id(4)	system identification (hostname) file	
ttytype(4)	data base of terminal types by port	
xfs(4)	layout of the XFS filesystem	

Table iii Miscellaneous Facilities

Reference Page	Function	Page
availmon(5)	overview of system availability monitoring facilities	

Table iv Protocols and Special Files

Reference Page	Function	Page
dks(7M)	dksc (SCSI) disk driver	
intro(7)	introduction to special files	

Table iv (continued) Protocols and Special Files

Reference Page	Function	Page
jag(7M)	dksc (SCSI) disk driver	
jagtape(7M)	SCSI tape interface	
keyboard(7)	keyboard specifications	
mouse(7)	mouse specifications	
mtio(7)	magnetic tape interface	
pcmouse(7)	mouse specifications	
root(7)	partition names	
rroot(7)	partition names	
rswap(7)	partition names	
rusr(7)	partition names	
serial(7)	serial communication ports	
swap(7)	partition names	
tps(7M)	SCSI tape interface	
tpsc(7M)	SCSI tape interface	
usr(7)	partition names	

NAME

Add_disk – add an optional disk to the system

SYNOPSIS

Add_disk [*controller_number*] [*disk_number*] [*lun_number*]

DESCRIPTION

Add_disk enables you to add an extra SCSI disk to a system if the disk is on an integral SCSI controller (i.e., it can not be used for disks attached to VME SCSI controllers).

The *disk_number* option must be specified if you are not adding the default ID of 2; similarly the controller and lun must be specified if other than 0.

The *Add_disk* command creates the required directory, makes the appropriate device file links, makes a new filesystem, does the required mount operation, and adds the appropriate entry to */etc/fstab*.

Appropriate checks are made for filesystems already existing on the common partitions (0, 6, and 7). If they are present, you are asked if you want to proceed before a filesystem is made. If the answer is no, *Add_disk* exits.

NOTE

Older versions of this command worked only with controller 0, and used a default mount point of */disk#*, where # was the SCSI ID. This version uses */disk##*, where the first # is the controller and the second is the SCSI ID.

Add_disk is a shell script and can be used as a template to determine what is necessary. The volume header on the disk must already have been initialized with the *fx*(1M) program.

SEE ALSO

fx(1M), mkfs(1M), fstab(4).

NAME

amreport – produce statistical and availability reports from availmon logfile

SYNOPSIS

/var/adm/avail/amreport [{ **–1** | **–l** *local_logfile* | **–s** *site_logfile* }]
[**–c**] [**–p**]

DESCRIPTION

amreport produces reports from the information gathered by the *availmon*(5) facility. The source of the information can be either a local logfile, specified by **–l**, or a site aggregate logfile, specified by **–s**. Without any argument, the command is equivalent to **amreport -l /var/adm/avail/availlog**. The options are:

–l *locallogfile*
Specify local logfile.

–s *sitelogfile*
Specify site logfile.

–1 Take one availability report as input from stdin and print it out in a more readable format.

–c Exclude current epoch in the reports.

–p Print the reports without using curses (for piping output or redirecting to a file).

amreport uses curses for screen control (unless **–p** is specified). It displays availability statistics and individual reports hierarchically from overall statistics for all systems, a table of statistics for all systems (if the input is a local log file, the above information is not provided), system-wise statistics for each system, a table of all reboot instances for each system, to availability reports (epochs) for each system. The control commands are shown at the bottom of each screen. In all types of records and tables, time instants are displayed in the format given by *ctime*(3C), and time durations are displayed in minutes, with equivalent days/hours/minutes in parentheses.

An epoch record contains the time at which the system was previously booted (which starts the epoch), the time it was stopped (ending the epoch), the reason for the stop, and the time it was rebooted. If the system stopped as a result of a hang, the exact instant at which it stopped is not easily known. In case of a system hang, the displayed uptime is estimated by a ticker daemon normally accurate to within 5 minutes on the lesser side of the actual uptime. If the ticker daemon is not enabled (see *amconfig*(1M)), the down time is assumed to be 1 minute.

System statistics contain the aggregation of epochs for the given system. The summary contains the total number of shutdowns of each kind; the average, minimum and maximum uptimes; the computed availability (see below); and the time at which system logging started (this is the beginning of the epoch during which installation of availmon was first performed).

In addition to epochs that were actually logged, it is assumed that the system is in the middle of a "current" epoch. Availability is computed based on logged epochs and the current epoch. If **−c** is specified, the current epoch is not considered in computing statistics.

The overall statistics (for an aggregate site log file) are similar to system statistics, except that the overall summary is the aggregation of all reports. The minimum and maximum uptime values include the names of the systems at which the corresponding times were recorded.

FILES

`/var/adm/avail/availlog`
local log file of availmon

SEE ALSO

amnotify(1M), amparse(1M), amreceive(1M), amregister(1M), amreport(1M), amsend(1M), amtickerd(1M), availmon(5).

NAME

autoconfig – configure kernel

SYNOPSIS

/etc/autoconfig [**–vnf**] [**–p** toolroot] [**–d** /var/sysgen]
[**–o** lbootopts] [**start**|**stop**]

DESCRIPTION

The *autoconfig* command is used invoke *lboot* and other commands to generate a UNIX kernel. The *autoconfig* command is also a startup script in */etc/init.d*.

The options are as:

–v Requests verbose output from *lboot* and other commands.

–f Generates a new kernel even if it appears that no hardware or software changes have been made.

–p *toolroot*
Specifies the directory tree containing the compiler and other tools needed to generate the kernel.

–d */var/sysgen*
Specifies the directory tree containing the system configuration modules and binaries.

–n Performs a dry run of *lboot* and reports whether a new kernel would be created or not. If **–f** is also given, it overrides this option.

start Used by *rc2* when the system is starting.

stop Used by *rc0* when the system is stopping.

The *autoconfig* command also uses the */var/config/autoconfig.options* file to tell *lboot* to configure a new kernel automatically or to prompt for permission before configuring a new kernel. The */var/config/autoconfig.options* file contains a **–T** by default, which indicates to *lboot* to configure the kernel automatically if necessary. This option can be changed to a **–t** to force *lboot* to prompt for permission before configuring a new kernel.

ENVIRONMENT VARIABLES

In addition to the environment variables used by *lboot*, *autoconfig* itself uses some environment variables. If you have these variables set for some other purpose, you may need to unset them before running *autoconfig*.

NOTE: This means that they should not be set in the global shell startup files in */etc*.

UNIX The file to check to see if it is out of date (defaults to */unix*) and also what the basename of the newly built kernel will be, if necessary.

SYSGEN Passed as the base directory for the kernel files (see also WORKDIR below) and as the base directory name for the arguments below, if they are not set in the environment.

BOOTAREA Passed as the **−b** argument to *lboot*.

SYSTEM Passed as the **−s** argument to *lboot*.

MASTERD Passed as the **−m** argument to *lboot*.

STUNEFILE Passed as the **−c** argument to *lboot*.

MTUNEDIR Passed as the **−n** argument to *lboot*.

WORKDIR Passed as the **−w** argument to *lboot*.

SEE ALSO

lboot(1M), rc0(1M), rc2(1M), setsym(1M).

NAME

Backup – backup the specified file or directory

SYNOPSIS

Backup [**–h** *hostname*] [**–i**] [**–t** *tapedevice*]
[*directory_name* | *file_name*]

DESCRIPTION

The *Backup* command archives the named file or directory (the current directory if none is specified) to the local or remote tape device. It can be used to make a full system backup by specifying the directory name as **/**.

In case of a full backup, this command makes a list of the files in the disk volume header and saves this information in a file which is then stored on tape. This file is used during crash recovery to restore a damaged volume header. The current date is saved in the file */etc/lastbackup*.

The options and arguments to *Backup* are:

–h *hostname* — If a tape drive attached to a remote host is used for backup, specify the name of the remote host with the **–h** *hostname* option. For remote backup to successfully work, you should have a TCP/IP network connection to the remote host and **guest** login privileges on that host.

–i — If a backup of all files modified since the date specified in the */etc/lastbackup* file is desired, specify the **–i** option. This option is valid only when doing a complete backup.

–t *tapedevice* — If the local or remote tape device is pointed to by a device file other than */dev/tape*, specify the device with the **–t** *tapedevice* option.

directory_name — Create a backup of the directory *directory_name*.

file_name — Create a backup of the file *file_name*.

The *Backup* command uses *bru*(1) to perform the backup function.

FILES

/tmp/volhdrlist — contains the list of the root volume header files
/etc/lastbackup — contains the date of last full backup

SEE ALSO

List_tape(1), Restore(1), bru(1).

NAME

bootp – server for Internet Bootstrap Protocol

SYNOPSIS

/usr/etc/bootp [**–d**] [**–f**]

DESCRIPTION

bootp is a server that supports the Internet 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 is 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:

   requester's network hardware address
   requester's Internet address (optional)
   desired server's name (optional)
   boot filename (optional)

2. All the BOOTP servers on the same network as the client machine receive the client's request. If the client has specified a particular server, then only that server responds.

3. The server looks up the requester in its configuration file by Internet address or network hardware address, in that order of preference. (The BOOTP configuration file is described below.) If the Internet address was not specified by the requester and a configuration record is not found, the server looks in the */etc/ethers* file (see *ethers*(4)) for an entry with the client's network hardware address. If an entry is found, the server checks the hostname of that entry against the */etc/hosts* file (see *hosts*(4)) in order to complete the network hardware 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 network hardware address into an Internet address by either of the two methods described, the server does 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 sends a response packet to the requester that includes the following information:

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

   If the boot file is missing, the server returns 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 requester, followed by a period and the requester's hostname. If that file is not present, remove the trailing period and hostname 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. Note that *tftpd*(1M) must be configured to allow access to the boot file (see the *tftpd*(1M) reference page for details).

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, ''Booting through Gateways,'' for an explanation.

Bootp Configuration File

In order to perform its name translation and address resolution functions, *bootp* requires configuration information that 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

IRIS   1  01:02:03:8a:8b:8c   192.0.2.1      unix
```

The fields of each line can 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 nonrooted pathname. The second section of the file is used for mapping client network hardware addresses into Internet addresses. Up to 512 hosts can be specified. The *htype* field should always have a value of 1 for now; this 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 six hexadecimal bytes separated by colons. The *iaddr* field is the 32-bit Internet address of the system expressed in standard Internet dot notation (see *inetd*(3N)). 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 selects the first of the following that it finds:

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

The length of the boot filename must not exceed 127 characters.

It is not necessary to create a record for every potential client in the *bootptab* file. The only constraint is that *bootp* responds only to a request from a client if it can deduce the client's Internet address. There are three ways that this can happen:

- The client already knows its Internet address and includes it in the BOOTP request packet.
- There is an entry in */usr/etc/bootptab* that matches the client's network hardware address.
- There are entries in the */etc/ethers* and */etc/hosts* files (or their NIS equivalents) that allow the client's network hardware address to be translated into an Internet address.

Booting through Gateways

Since the BOOTP request is distributed using a UDP broadcast, it is only received by other hosts on the same network 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 network. To use BOOTP forwarding, there must be a *bootp* process running in a gateway machine on the local network. A gateway machine is simply a machine with more than one network interface board. The gateway *bootp* must be invoked with the **–f** option to activate forwarding. Such a forwarding *bootp* resends 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 is automatically routed to the requested BOOTP server if the kernel routing tables contain a route to the destination network.

DIAGNOSTICS

The BOOTP server logs messages using 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 failed (`*message*`)`
`forwarding failed (`*message*`)`
`send failed (`*message*`)`
`set arp ioctl failed`
: Each of the above messages means that a system call has returned an error unexpectedly. Such errors usually cause *bootp* to terminate. The *message* is the appropriate standard system error message.

`less than two interfaces, –f flag ignored`
: Warning only (debug mode). 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 can be given as a hostname or an Internet address.

`request from` *xxx* `for 'fff'`
: Information only. *bootp* logs each request for a boot file. The host *xxx* has requested boot file *fff*.

`can't access boot file fff (`*message*`)`
: A request has been received for the boot file *fff*, but that file is not accessible.

`reply boot filename fff too long`
: The filename length *fff* exceeds the BOOTP protocol limit of 127 characters.

`reply boot file fff`
: Information only. *bootp* has selected the file *fff* as the boot file to satisfy a request.

`can't reply to dd.dd.dd.dd (unknown net)`
: This *bootp* has generated a response to a client and is trying to send the response directly to the client (that is, the request did not get forwarded by another *bootp*), but none of the network interfaces on this machine is on the same directly connected network as the client machine.

`reply: can't find net for dd.dd.dd.dd`
The server is acting as BOOTP forwarder and has received a datagram with a client address that is not on a directly connected network.

`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 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*(3C) failed.

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

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

SEE ALSO

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

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 filesystem. If the root filesystem is found to be bad, *bcheckrc* repairs it.

Then, the *brc* procedure clears the mounted filesystem table, */etc/mtab*, and puts the entry for the root filesystem 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 is placed in the multiuser state via the */etc/rc2* procedure.

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

SEE ALSO

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

NAME

chkconfig – configuration state checker

SYNOPSIS

chkconfig [**–s**]
chkconfig *flag*
chkconfig [**–f**] *flag* [**on** | **off**]

DESCRIPTION

chkconfig with no arguments or with the **–s** option prints the state (**on** or **off**) of every configuration flag found in the directory */var/config*. The flags normally are shown sorted by name; with the **–s** option they are shown sorted by state.

A flag is considered **on** if its file contains the string ''on'' and **off** otherwise.

If *flag* is specified as the sole argument, *chkconfig* exits with status 0 if *flag* is **on** and with status 1 if *flag* is **off** or nonexistent. The exit status can be used by shell scripts to test the state of a flag. Here is an example using *sh*(1) syntax:

```
if /sbin/chkconfig verbose; then
     echo "Verbose is on"
else
     echo "Verbose is off"
fi
```

The optional third argument allows the specified flag to be set. The flag file must exist in order to change its state. Use the **–f** (''force'') option to create the file if necessary.

These flags are used for determining the configuration status of the various available subsystems and daemons during system startup and during system operation.

A daemon or subsystem is enabled if its configuration flag in the */var/config* directory is in the **on** state. If the flag file is missing, the flag is considered **off**. The following is a list of available flags and the associated action if the flag is **on**. Depending upon your configuration, they may not all be available on your system.

4DDN Initialize 4DDN (DECnet connectivity) software.

acct Start process accounting.

automount Start the NFS automounter daemon.

desktop If off, fewer of the Indigo Magic user interface features are enabled, and typically a different toolchest menu is used. It is identical to creating the file *$HOME/.disableDesktop* except that it applies to all accounts. The specific effect is that the desktop version of Xsession (*/usr/lib/X11/xdm/Xsession.dt*) is not run upon login, and therefore programs

started from that file are not run or are run with different options.

directoryserver Start the Cadmin directory server daemon.

gated Start the Cornell routing daemon instead of the BSD routed.

hypernet Initialize HyperNET controller and routes.

ipfilterd Enable the Silicon Graphics IP Packet Filtering daemon.

jserver Start Japanese convert engine if the optional product *Japanese Language Module* is installed.

lockd Start the NFS lock and status daemons.

mediad Start the removable media daemon.

mrouted Start the IP multicast routing daemon (useful only on gateways).

named Start Internet domain name server.

network Allow incoming and outgoing network traffic. This flag can be set off if you need to isolate the machine from the network without removing cables.

nfs Start the NFS daemons *nfsd* and *biod*. Mount all NFS filesystems.

noiconlogin Do not show user icons on the login screen.

nsr Start up the IRIS NetWorker daemons. See *nsr*(1M) for more details.

nostickytmp Do not turn the sticky bit on for the directories */tmp* and */var/tmp*.

objectserver Start the Cadmin object server daemon.

pcnfsd Start the PC-NFS server daemon.

quotacheck Run *quotacheck*(1M) on the filesystems that have quotas enabled. See *quotas*(4) for more details.

quotas Enable quotas for local configured filesystems.

rarpd Start the Reverse ARP daemon.

routed Start the 4.3BSD RIP routing daemon. See *routed*(1M) for more details.

rsvpd Start the RSVP daemon. See *rsvpd*(1M) for more details.

rtnetd Initialize preemptable networking for real-time use.

rwhod Start the 4.3BSD rwho daemon.

sar Start the system activity reporter.

snmpd Start the Simple Network Management Protocol daemon.

soundscheme Start the Indigo Magic audio cue daemon.

timed Start the 4.3BSD time synchronization daemon.

timeslave Start the Silicon Graphics time synchronization daemon.

verbose Print the names of daemons as they are started.

vswap Add virtual swap. See *swap*(1M) for a discussion of virtual swap. By default 80000 blocks are added. You can increase or decrease this amount by modifying the */var/config/vswap.options* file.

visuallogin Enable the visual login screen.

windowsystem Start the X window system. If **windowsystem** is **off**, it is necessary to modify the *inittab*(4) file to enable *getty*(1M) on the textport window if you wish to use graphics as a dumb terminal.

The recommended means of enabling and disabling the window system are the commands *startgfx*(1G) and *stopgfx*(1G).

xdm Start the X display manager.

yp Enable NIS, start the ypbind daemon.

ypmaster If **yp** is **on**, become the NIS master and start the passwd server. The **ypserv** flag should be on too.

ypserv If **yp** is **on**, become a NIS server.

FILES

/var/config directory containing configuration flag files

SEE ALSO

cron(1M), rc0(1M), rc2(1M).

NAME

chroot – change root directory for a command

SYNOPSIS

chroot newroot command

DESCRIPTION

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

If you redirect the output of the command to a file:

chroot *newroot command* **>** *x*

chroot creates the file *x* relative to the original root of the command, not the new one.

The new root pathname 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 superuser.

CAVEAT

In order to execute programs that use shared libraries, the following directories and their contents must be present in the new root directory.

./lib This directory must contain the run-time loader (*/lib/rld*) and any shared object files needed by your applications (usually including *libc.so.1*). That means it must normally be in */lib* and a symlink in */usr/lib* to *../../lib/libc.so.1P*.

./dev The run-time loader needs the zero device in order to work correctly. Copy */dev/zero* into this directory and make it readonly (mode 444).

SEE ALSO

cd(1), chroot(2), ftpd(1m) (for more comments on issues in setting up chroot'ed environments)

NOTES

Exercise extreme caution when referencing device files in the new root filesystem.

When using *chroot*, with commands that are dynmically linked, all of the libraries required must be in the chroot'ed environment. The system will usually log a message in */var/adm/SYSLOG* if some libraries or *rld* are not found.

NAME

csh – shell command interpreter with a C-like syntax

SYNOPSIS

csh [–**bcefinstvVxX**] [*argument* ...]

DESCRIPTION

csh, the C shell, is a command interpreter with a syntax reminiscent of the C language. It provides a number of convenient features for interactive use that are not available with the standard (Bourne) shell, including filename completion, command aliasing, history substitution, job control, and a number of built-in commands. As with the standard shell, the C shell provides variable, command and filename substitution.

Initialization and Termination

When first started, the C shell normally performs commands from the **.cshrc** file in your home directory, provided that it is readable and you either own it or your real group ID matches its group ID. If the shell is invoked with a name that starts with '–', as when started by **login**(1), the shell runs as a **login** shell. In this case, before executing the commands from the **.cshrc** file, the shell executes the commands from the following files in the order specified: **/etc/cshrc**, **/etc/.login** and **/etc/csh.cshrc**. These files can be used to provide system-wide settings for all **csh** users. After executing commands from the **.cshrc** file, a login shell executes commands from the **.login** file in your home directory; the same permission checks as those for **.cshrc** are applied to this file. Typically, the **.login** file contains commands to specify the terminal type and environment. Please note that **csh** can run as a **login** shell if it is invoked upon startup of a window shell such as **xwsh**(1G). This is so any terminal type information that might be contained in the **.login** file(s) can be made known to the window shell.

As a login shell terminates, it performs commands from the **.logout** file in your home directory; the same permission checks as those for **.cshrc** are applied to this file.

Interactive Operation

After startup processing is complete, an interactive C shell begins reading commands from the terminal, prompting with *hostname***%** (or *hostname***#** for the privileged user). The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the history list and then parsed, as described under USAGE, below. Finally, the shell executes each command in the current line.

Noninteractive Operation

When running noninteractively, the shell does not prompt for input from the terminal. A noninteractive C shell can execute a command supplied as an *argument* on its command line, or interpret commands from a script.

The following options are available:

−b Force a "break" from option processing. Subsequent command-line arguments are not interpreted as C shell options. This allows the passing of options to a script without confusion. The shell does not run a set-user-ID script unless this option is present.

−c Read commands from the first filename *argument* (which must be present). Remaining arguments are placed in **argv**, the argument-list variable.

−e Exit if a command terminates abnormally or yields a nonzero exit status.

−f Fast start. Read neither the **.cshrc** file, nor the **.login** file (if a login shell) upon startup.

−i Forced interactive. Prompt for command line input, even if the standard input does not appear to be a terminal (character-special device).

−n Parse (interpret), but do not execute commands. This option can be used to check C shell scripts for syntax errors.

−s Take commands from the standard input.

−t Read and execute a single command line. A '\' (backslash) can be used to escape each newline for continuation of the command line onto subsequent input lines.

−v Verbose. Set the **verbose** predefined variable; command input is echoed after history substitution (but before other substitutions) and before execution.

−V Set **verbose** before reading **.cshrc**.

−x Echo. Set the **echo** variable; echo commands after all substitutions and just before execution.

−X Set **echo** before reading **.cshrc**.

Except with the options **−c**, **−i**, **−s**, or **−t,** the first nonoption *argument* is taken to be the name of a command or script. It is passed as argument zero, and subsequent arguments are added to the argument list for that command or script. **csh** scripts should always start with the line

```
#! /bin/csh −f
```

which causes the script to be executed by **/bin/csh** even if invoked by a user running a shell other than **csh** and inhibits processing of the **.cshrc** file to prevent interference from aliases defined by the invoking user.

USAGE

Filename Completion

When enabled by setting the variable **filec**, an interactive C shell can complete a partially typed filename or user name. When an unambiguous partial filename is followed by an ESC character on the terminal input line, the shell fills in the remaining characters of a matching filename from the working directory.

If a partial filename is followed by the EOF character (usually typed as <Ctrl-d>), the shell lists all filenames that match. It then prompts once again, supplying the incomplete command line typed in so far.

When the last (partial) word begins with a tilde (~), the shell attempts completion with a user name, rather than a file in the working directory.

The terminal bell signals errors or multiple matches; this can be inhibited by setting the variable **nobeep**. You can exclude files with certain suffixes by listing those suffixes in the variable **fignore**. If, however, the only possible completion includes a suffix in the list, it is not ignored. **fignore** does not affect the listing of filenames by the EOF character.

Lexical Structure

The shell splits input lines into words at space and tab characters, except as noted below. The characters **&**, |, **;**, **<**, **>**, **(**, and **)** form separate words; if paired, the pairs form single words. These shell metacharacters can be made part of other words, and their special meaning can be suppressed by preceding them with a '\' (backslash). A newline preceded by a **** is equivalent to a space character.

In addition, a string enclosed in matched pairs of single-quotes (´), double-quotes (**"**), or backquotes (`), forms a partial word; metacharacters in such a string, including any space or tab characters, do not form separate words. Within pairs of backquote (`**)** or double-quote (**"**) characters, a newline preceded by a '\' (backslash) gives a true newline character. Additional functions of each type of quote are described, below, under **Variable Substitution**, **Command Substitution**, and **Filename Substitution**.

When the shell's input is not a terminal, the character **#** introduces a comment that continues to the end of the input line. Its special meaning is suppressed when preceded by a **** or enclosed in matching quotes.

Command Line Parsing

A *simple command* is composed of a sequence of words. The first word (that is not part of an I/O redirection) specifies the command to be executed. A simple command, or a set of simple commands separated by **|** or **|&** characters, forms a *pipeline*. With **|**, the standard output of the preceding command is redirected to the standard input of the command that follows. With **|&**, both the standard error and the standard output are redirected through the pipeline.

Pipelines can be separated by semicolons (**;**), in which case they are executed sequentially. Pipelines that are separated by **&&** or **||** form conditional sequences in which the execution of pipelines on the right depends upon the success or failure, respectively, of the pipeline on the left.

A pipeline or sequence can be enclosed within parentheses '**( )**' to form a simple command that can be a component in a pipeline or sequence.

A sequence of pipelines can be executed asynchronously, or "in the background" by appending an '**&**'; rather than waiting for the sequence to finish before issuing a prompt, the shell displays the job number (see **Job Control**, below) and associated process IDs, and prompts immediately.

History Substitution

History substitution allows you to use words from previous command lines in the command line you are typing. This simplifies spelling corrections and the repetition of complicated commands or arguments. Command lines are saved in the history list, the size of which is controlled by the **history** variable. The most recent command is retained in any case. A history substitution begins with a **!** (although you can change this with the **histchars** variable) and can occur anywhere on the command line; history substitutions do not nest. The **!** can be escaped with **** to suppress its special meaning.

Input lines containing history substitutions are echoed on the terminal after being expanded, but before any other substitutions take place or the command gets executed.

Event Designators

An event designator is a reference to a command-line entry in the history list.

! Start a history substitution, except when followed by a space character, tab, newline, **=** or **(**.

!! Refer to the previous command. By itself, this substitution repeats the previous command.

!*n* Refer to command line *n* .

!–*n* Refer to the current command line minus *n*.

!str Refer to the most recent command starting with **str**.

!?*str*[**?**] Refer to the most recent command containing **str**.

!{...**}** Insulate a history reference from adjacent characters (if necessary).

Word Designators

A '**:**' (colon) separates the event specification from the word designator. It can be omitted if the word designator begins with a **^**, **$**, *****, – or **%**. If the word is to be selected from the previous command, the second **!** character can be omitted from the event specification. For instance, **!!:1** and **!:1** both refer to the first word of the previous command, while **!!$** and **!$** both refer to the last word in the previous command. Word designators include:

The entire command line typed so far.

0 The first input word (command).
n The *n*'th argument.
^ The first argument, that is, **1**.
$ The last argument.
% The word matched by (the most recent) **?***s* search.
x–*y* A range of words; –*y* abbreviates **0**–*y*.
***** All the arguments, or a null value if there is just one word in the event.
*x****** Abbreviates *x*–**$**.
x– Like *x** but omitting word **$**.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a **:**.

h Remove a trailing pathname component, leaving the head.
r Remove a trailing suffix of the form '**.***xxx*', leaving the basename.
e Remove all but the suffix.
s/*l***/***r*[**/**]
Substitute *r* for *l*.
t Remove all leading pathname components, leaving the tail.
& Repeat the previous substitution.
g Apply the change to the first occurrence of a match in each word, by prefixing the above (for example, **g&**).
p Print the new command but do not execute it.
q Quote the substituted words, escaping further substitutions.
x Like **q**, but break into words at each space character, tab or newline.

Unless preceded by a **g**, the modification is applied only to the first string that matches *l*; an error results if no string matches.

The left-hand side of substitutions are not regular expressions, but character strings. Any character can be used as the delimiter in place of **/**. A backslash quotes the delimiter character. The character **&**, in the right hand side, is replaced by the text from the left-hand-side. The **&** can be quoted with a backslash. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* from **!?***s*. You can omit the rightmost delimiter if a newline immediately follows *r*; the rightmost **?** in a context scan can similarly be omitted.

Without an event specification, a history reference refers either to the previous command, or to a previous history reference on the command line (if any).

Quick Substitution

^*l*^*r*[^] This is equivalent to the history substitution: **!:s**^*l*^*r*[^].

Aliases

The C shell maintains a list of aliases that you can create, display, and modify using the **alias** and **unalias** commands. The shell checks the first word in each command to see if it matches the name of an existing alias. If it does, the command is reprocessed with the alias definition replacing its name; the history substitution mechanism is made available as though that command were the previous input line. This allows history substitutions, escaped with a backslash in the definition, to be replaced with actual command-line arguments when the alias is used. If no history substitution is called for, the arguments remain unchanged.

Aliases can be nested. That is, an alias definition can contain the name of another alias. Nested aliases are expanded before any history substitutions is applied. This is useful in pipelines such as

```
alias lm ´ls –l \!* | more´
```

which when called, pipes the output of **ls**(1) through **more**(1).

Except for the first word, the name of the alias cannot appear in its definition, nor in any alias referred to by its definition. Such loops are detected, and cause an error message.

I/O Redirection

The following metacharacters indicate that the subsequent word is the name of a file to which the command's standard input, standard output, or standard error is redirected; this word is variable, command, and filename expanded separately from the rest of the command.

< Redirect the standard input.

<<*word* Read the standard input, up to a line that is identical with *word*, and place the resulting lines in a temporary file. Unless *word* is escaped or quoted, variable and command substitutions are performed on these lines. Then, invoke the pipeline with the temporary file as its standard input. *word* is not subjected to variable, filename, or command substitution, and each line is compared to it before any substitutions are performed by the shell.

> >! >& >&!
Redirect the standard output to a file. If the file does not exist, it is created. If it does exist, it is overwritten; its previous contents are lost.

When set, the variable **noclobber** prevents destruction of existing files. It also prevents redirection to terminals and **/dev/null**, unless one of the **!** forms is used. The **&** forms redirect both standard output and the standard error (diagnostic output) to the file.

>> >>& >>! >>&!

Append the standard output. Like **>**, but places output at the end of the file rather than overwriting it. If **noclobber** is set, it is an error for the file not to exist, unless one of the **!** forms is used. The **&** forms append both the standard error and standard output to the file.

Variable Substitution

The C shell maintains a set of *variables*, each of which is composed of a *name* and a *value*. A variable name consists of up to 20 letters and digits, and starts with a letter (the underscore is considered a letter). A variable's value is a space-separated list of zero or more words.

To refer to a variable's value, precede its name with a '**$**'. Certain references (described below) can be used to select specific words from the value, or to display other information about the variable. Braces can be used to insulate the reference from other characters in an input-line word.

Variable substitution takes place after the input line is analyzed, aliases are resolved, and I/O redirections are applied. Exceptions to this are variable references in I/O redirections (substituted at the time the redirection is made), and backquoted strings (see Command Substitution).

Variable substitution can be suppressed by preceding the **$** with a \, except within double-quotes where it always occurs. Variable substitution is suppressed inside of single-quotes. A **$** is escaped if followed by a space character, tab or newline.

Variables can be created, displayed, or destroyed using the **set** and **unset** commands. Some variables are maintained or used by the shell. For instance, the **argv** variable contains an image of the shell's argument list. Of the variables used by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not.

Numerical values can be operated on as numbers (as with the **@** built-in). With numeric operations, an empty value is considered to be zero; the second and subsequent words of multiword values are ignored. For instance, when the **verbose** variable is set to any value (including an empty value), command input is echoed on the terminal.

Command and filename substitution is subsequently applied to the words that result from the variable substitution, except when suppressed by double-quotes, when **noglob** is set (suppressing filename substitution), or when the reference is quoted with the **:q** modifier. Within double-quotes, a reference is expanded to form (a portion of) a quoted string; multiword values are expanded to a string with embedded space characters. When the **:q** modifier is applied to the reference, it is expanded to a list of space-separated words, each of which is quoted to prevent subsequent command or filename substitutions.

Except as noted below, it is an error to refer to a variable that is not set.

$*var*

${*var***}** These are replaced by words from the value of *var*, each separated by a space character. If *var* is an environment variable, its value is returned (but '**:**' modifiers and the other forms given below are not available).

$*var*[*index*]
${***var***[***index***]}** These select only the indicated words from the value of *var*. Variable substitution is applied to *index*, which can consist of (or result in) a either single number, two numbers separated by a '–', or an asterisk. Words are indexed starting from 1; a '*****' selects all words. If the first number of a range is omitted (as with **$argv[–2]), it defaults to 1. If the last number of a range is omitted (as with **$argv[1–]**), it defaults to **$#***var* (the word count). It is not an error for a range to be empty if the second argument is omitted (or within range).

$#*name*
${#*name***}** These give the number of words in the variable.

$0 This substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$*n*
${***n***}** Equivalent to **$argv[*n***]**.

$*** Equivalent to **$argv[*].

The modifiers **:e**, **:h**, **:q**, **:r**, **:t** and **:x** can be applied (see **History Substitution**), as can **:gh**, **:gt** and **:gr**. If **{}** (braces) are used, then the modifiers must appear within the braces. The current implementation allows only one such modifier per expansion.

The following references cannot be modified with **:** modifiers.

$?*var*
${?*var***}** Substitutes the string 1 if *var* is set or 0 if it is not set.

$?0 Substitutes 1 if the current input filename is known, or 0 if it is not.

$$ Substitute the process number of the (parent) shell.

$< Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a C shell script.

Command and Filename Substitutions

Command and filename substitutions are applied selectively to the arguments of built-in commands. Portions of expressions that are not evaluated are not expanded. For non-built-in commands, filename expansion of the command name is done separately from that of the argument list; expansion occurs in a subshell, after I/O redirection is performed.

Command Substitution

A command enclosed by backquotes (`...`) is performed by a subshell. Its standard output is broken into separate words at each space character, tab and newline; null words are discarded. This text replaces the backquoted string on the current command line. Within double-quotes, only newline characters force new words; space and tab characters are preserved. However, a final newline is ignored. It is therefore possible for a command substitution to yield a partial word.

Filename Substitution

Unquoted words containing any of the characters *****, **?**, **[** or **{**, or that begin with **~**, are expanded (also known as *globbing*) to an alphabetically sorted list of filenames, as follows:

*****	Match any (zero or more) characters.
?	Match any single character.
[... **]**	Match any single character in the enclosed list(s) or range(s). A list is a string of characters. A range is two characters separated by a minus-sign (–), and includes all the characters in between in the ASCII collating sequence (see **ascii**(5)).
{ *str*, *str*, ... **}**	Expand to each string (or filename-matching pattern) in the comma-separated list. Unlike the pattern-matching expressions above, the expansion of this construct is not sorted. For instance, **{b,a}** expands to '**b**' '**a**', (not '**a**' '**b**'). As special cases, the characters **{** and **}**, along with the string **{}**, are passed undisturbed.
~[*user*]	Your home directory, as indicated by the value of the variable **home**, or that of *user*, as indicated by the password entry for *user*.

Only the patterns *****, **?** and **[**...**]** imply pattern matching; an error results if no filename matches a pattern that contains them. The '**.**' (dot character), when it is the first character in a filename or pathname component, must be matched explicitly. The **/** (slash) must also be matched explicitly.

Expressions and Operators

A number of C shell built-in commands accept expressions, in which the operators are similar to those of C and have the same precedence. These expressions typically appear in the **@**, **exit**, **if**, **set** and **while** commands, and are often used to regulate the flow of control for executing commands. Components of an expression are separated by white space.

Null or missing values are considered 0. The result of all expressions are strings, which can represent decimal numbers.

The following C shell operators are grouped in order of precedence:

(…**)**	grouping
~	one's complement
!	logical negation
*** / %**	multiplication, division, remainder (These are right associative, which can lead to unexpected results. Group combinations explicitly with parentheses.)
+ –	addition, subtraction (also right associative)
<< >>	bitwise shift left, bitwise shift right
< > <= >=	less than, greater than, less than or equal to, greater than or equal to
== != =~ !~	equal to, not equal to, filename-substitution pattern match (described below), filename-substitution pattern mismatch
&	bitwise AND
^	bitwise XOR (exclusive or)
|	bitwise inclusive OR
&&	logical AND
||	logical OR

The operators: **==**, **!=**, **=~**, and **!~** compare their arguments as strings; other operators use numbers. The operators **=~** and **!~** each check whether or not a string to the left matches a filename substitution pattern on the right. This reduces the need for **switch** statements when pattern-matching between strings is all that is required.

Also available are file inquiries:

–**r** *file*	Return true, or 1 if the user has read access. Otherwise it returns false, or 0.
–**w** *file*	True if the user has write access.
–**x** *file*	True if the user has execute permission (or search permission on a directory).
–**e** *file*	True if *file* exists.
–**o** *file*	True if the user owns *file*.
–**z** *file*	True if *file* is of zero length (empty).
–**f** *file*	True if *file* is a plain file.
–**d** *file*	True if *file* is a directory.
–**l** *file*	True if *file* is a symbolic link.
–**c** *file*	True if *file* is a character special file.
–**b** *file*	True if *file* is a block special file.
–**p** *file*	True if *file* is a named pipe (fifo).
–**u** *file*	True if *file* has the set-user-ID permission bit set (see *chmod*(1)).
–**g** *file*	True if *file* has the set-group-ID permission bit set (see *chmod*(1)).
–**k** *file*	True if *file* has the sticky bit set (see *chmod*(1)).
–**s** *file*	True if *file* has size strictly greater than zero.

−t *file* True if *file* is an open file descriptor for a terminal device.

If *file* does not exist or is inaccessible, then all inquiries return false.

An inquiry as to the success of a command is also available:

{ *command* **}** If *command* runs successfully, the expression evaluates to true, 1. Otherwise it evaluates to false 0. (Note that, conversely, *command* itself typically returns 0 when it runs successfully, or some other value if it encounters a problem. If you want to get at the status directly, use the value of the **status** variable rather than this expression).

Control Flow

The shell contains a number of commands to regulate the flow of control in scripts, and within limits, from the terminal. These commands operate by forcing the shell either to reread input (to *loop*), or to skip input under certain conditions (to *branch*).

Each occurrence of a **foreach**, **switch**, **while**, **if**...**then** and **else** built-in must appear as the first word on its own input line.

If the shell's input is not seekable and a loop is being read, that input is buffered. The shell performs seeks within the internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward **goto** commands succeed on nonseekable inputs.)

Command Execution

If the command is a C shell built-in, the shell executes it directly. Otherwise, the shell searches for a file by that name with execute access. If the command-name contains a **/**, the shell takes it as a pathname, and searches for it. If the command-name does not contain a **/**, the shell attempts to resolve it to a pathname, searching each directory in the **path** variable for the command. To speed the search, the shell uses its hash table (see the **rehash** built-in) to eliminate directories that have no applicable files. This hashing can be disabled with the **−c** or **−t**, options, or the **unhash** built-in.

As a special case, if there is no **/** in the name of the script and there is an alias for the word **shell**, the expansion of the **shell** alias is prepended (without modification), to the command line. The system attempts to execute the first word of this special (late-occurring) alias, which should be a full pathname. Remaining words of the alias's definition, along with the text of the input line, are treated as arguments.

When a pathname is found that has proper execute permissions, the shell forks a new process and passes it, along with its arguments to the kernel (using the **execve**(2) system call). The kernel then attempts to overlay the new process with the desired program. If the file is an executable binary (in **a.out**(4) format) the kernel succeeds, and begins executing the new process. If the file is a text file, and the first line begins with **#!**, the next word is taken to be the pathname of a shell (or command) to interpret that script. Subsequent words on the first line are taken as options for that shell. The kernel invokes (overlays) the indicated shell, using the name of the script as an argument.

If neither of the above conditions holds, the kernel cannot overlay the file (the **execve**(2) call fails); the C shell then attempts to execute the file by spawning a new shell, as follows:

- If the first character of the file is a **#**, a C shell is invoked.
- Otherwise, a standard (Bourne) shell is invoked.

Signal Handling

The shell normally ignores QUIT signals. Background jobs are immune to signals generated from the keyboard, including hangups (HUP). Other signals have the values that the C shell inherited from its environment. The shell's handling of interrupt and terminate signals within scripts can be controlled by the **onintr** built-in. Login shells catch the TERM signal; otherwise this signal is passed on to child processes. In no case are interrupts allowed when a login shell is reading the **.logout** file.

Job Control

The shell associates a numbered *job* with each command sequence, to keep track of those commands that are running in the background or have been stopped with TSTP signals (typically <Ctrl-z>). When a command, or command sequence (semicolon separated list), is started in the background using the **&** metacharacter, the shell displays a line with the job number in brackets, and a list of associated process numbers:

```
[1] 1234
```

To see the current list of jobs, use the **jobs** built-in command. The job most recently stopped (or put into the background if none are stopped) is referred to as the *current* job, and is indicated with a '**+**'. The previous job is indicated with a '–'; when the current job is terminated or moved to the foreground, this job takes its place (becomes the new current job).

To manipulate jobs, refer to the **bg**, **fg**, **kill**, **stop** and **%** built-ins.

A reference to a job begins with a '**%**'. By itself, the percent-sign refers to the current job.

% %+ %% The current job.

%– The previous job.

%*j* Refer to job *j* as in: '**kill –9 %***j*'. *j* can be a job number, or a string that uniquely specifies the command line by which it was started; '**fg %vi**' might bring a stopped **vi** job to the foreground, for instance.

%?*string* Specify the job for which the command line uniquely contains *string*.

A job running in the background stops when it attempts to read from the terminal. Background jobs can normally produce output, but this can be suppressed using the '**stty tostop**' command.

Status Reporting

While running interactively, the shell tracks the status of each job and reports whenever a finishes or becomes blocked. It normally displays a message to this effect as it issues a prompt, so as to avoid disturbing the appearance of your input. When set, the **notify** variable indicates that the shell is to report status changes immediately. By default, the **notify** command marks the current process; after starting a background job, type **notify** to mark it.

Built-In Commands

Built-in commands are executed within the C shell. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

: Null command. This command is interpreted, but performs no action.

alias [*name* [*def*]]
Assign *def* to the alias *name*. *def* is a list of words that can contain escaped history-substitution metasyntax. *name* is not allowed to be **alias** or **unalias**. If *def* is omitted, the alias *name* is displayed along with its current definition. If both *name* and *def* are omitted, all aliases are displayed.

bg [**%** Run the current or specified jobs in the background.

break Resume execution after the **end** of the nearest enclosing **foreach** or **while** loop. The remaining commands on the current line are executed. This allows multilevel breaks to be written as a list of **break** commands, all on one line.

breaksw Break from a **switch**, resuming after the **endsw**.

case *label***:** A label in a **switch** statement.

cd [*dir*]
chdir [*dir*] Change the shell's working directory to directory *dir*. If no argument is given, change to the home directory of the user. If *dir* is a relative pathname not found in the current directory, check for it in those directories listed in the **cdpath** variable. If *dir* is the name of a shell variable whose value starts with a **/**, change to the directory named by that value.

continue Continue execution of the nearest enclosing **while** or **foreach**.

default: Labels the default case in a **switch** statement. The default should come after all **case** labels. Any remaining commands on the command line are first executed.

dirs [**−l**] Print the directory stack, most recent to the left; the first directory shown is the current directory. With the **−l** argument, produce an unabbreviated printout; use of the **~** notation is suppressed.

echo [**−n**] *list* The words in *list* are written to the shell's standard output, separated by space characters. The output is terminated with a newline unless the **−n** option or the **\c** escape is specified. The following C-like escape sequences are available:

\b	backspace
\c	print line without newline
\f	formfeed
\n	newline
\r	carriage return
\t	tab
****	backslash
\0*n*	the 8-bit character whose code is the 1-, 2- or 3-digit octal number *n*. Note that *n* (no leading zero) is accepted for backwards compatibility with older IRIX cshs. This can cause unexpected results in older scripts if the character immediately trailing three digits is also numeric.

eval *argument ...*
Reads the arguments as input to the shell, and executes the resulting command(s). This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See **tset**(1) for an example of how to use **eval**.

exec *command* Execute *command* in place of the current shell, which terminates.

exit [**(***expr***)**] The shell exits, either with the value of the STATUS variable, or with the value of the specified by the expression **expr**.

fg % [*job*] Bring the current or specified *job* into the foreground.

foreach *var* **(***wordlist***)**
...
end
The variable *var* is successively set to each member of *wordlist*. The sequence of commands between this command and the matching **end** is executed for each new value of *var*. (Both **foreach** and **end** must appear alone on separate lines.)

The built-in command **continue** can be used to continue the loop prematurely and the built-in command **break** to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with **?** before any statements in the loop are executed.

glob *wordlist* Perform filename expansion on *wordlist*. Like **echo**, but no **** escapes are recognized. Words are delimited by **NULL** characters in the output.

goto *label* The specified *label* is filename and command expanded to yield a label. The shell rewinds its input as much as possible and searches for a line of the form *label***:** possibly preceded by space or tab characters. Execution continues after the indicated line. It is an error to jump to a label that occurs between a **while** or **for** built-in, and its corresponding **end**.

hashstat Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding **exec**s). An **exec** is attempted for each component of the *path* where the hash function indicates a possible hit, and in each component that does not begin with a '**/**'.

history [−**hr**] [*n*]
Display the history list; if *n* is given, display only the *n* most recent events.

−**r** Reverse the order of printout to be most recent first rather than oldest first.
−**h** Display the history list without leading numbers. This is used to produce files suitable for sourcing using the −**h** option to *source*.

if (*expr***)** *command*
If the specified expression evaluates to true, the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Note: I/O redirection occurs even if **expr** is false, when *command* is *not* executed (this is a bug).

if (*expr***) then**
...
else if (*expr2***) then**
...
else
...
endif If **expr**"" is true, commands up to the first **else** are executed. Otherwise, if *expr2* is true, the commands between the **else if** and the second **else** are executed. Otherwise, commands between the **else** and the **endif** are executed. Any number of **else if** pairs are allowed, but only one **else**. Only one **endif** is needed, but it is required. The words **else** and **endif** must be the first nonwhite characters on a line. The **if** must appear alone on its input line or after an **else**.)

jobs [−**l**] List the active jobs under job control.

−**l** List process IDs, in addition to the normal information.

kill [*–sig*] [*pid*] [**%***job*] ...

kill –l Send the TERM (terminate) signal, by default, or the signal specified, to the specified process ID, the *job* indicated, or the current *job*. Signals are either given by number or by name. There is no default. Typing **kill** does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process is sent a CONT (continue) signal as well.

–**l** List the signal names that can be sent.

limit [–**h**] [*resource* [*max-use*]]

Limit the consumption by the current process or any process it spawns, each not to exceed *max-use* on the specified *resource*. If *max-use* is omitted, print the current limit; if *resource* is omitted, display all limits.

–**h** Use hard limits instead of the current limits. Hard limits impose a ceiling on the values of the current limits. Only the privileged user can raise the hard limits.

resource is one of:

cputime	Maximum CPU seconds per process.
filesize	Largest single file allowed.
datasize	Maximum data size (including stack) for the process.
stacksize	Maximum stack size for the process. Note: If this is set too high, **sproc**(2) may fail.
coredumpsize	Maximum size of a core dump (file).
memoryuse	Maximum amount of physical memory per process (resident set size).
vmemoryuse	Maximum amount of virtual memory per process, including text, data, heap, shared memory, mapped files, stack, etc..
descriptors	Maximum number of open file descriptors per process.

max-use is a number, with an optional scaling factor, as follows:

*n***h**	Hours (for **cputime**).
*n***k**	*n* kilobytes. This is the default for all file or memory size limits.
*n***m**	*n* megabytes or minutes (for **cputime**).
*mm***:***ss**	Minutes and seconds (for **cputime**).

The *resource* argument can be abbreviated by using only enough characters to make the name unambiguous. Refer to the **setrlimit**(2) manual entry for more information about process resource limits.

login [*username* |–**p**]
Terminate a login shell and invoke **login**(1). The **.logout** file is not processed. If *username* is omitted, **login** prompts for the name of a user.

–**p** Preserve the current environment (variables).

logout Terminate a login shell.

nice [**+***n* |–*n*] [*command*]
Increment the process priority value for the shell or for *command* by *n*. The higher the priority value, the lower the priority of a process, and the slower it runs. When given, *command* is always run in a subshell, and the restrictions placed on commands in simple **if** commands apply. If *command* is omitted, **nice** increments the value for the current shell. If no increment is specified, **nice** sets the process priority value to 4. The range of process priority values is from –20 to 20. Values of *n* outside this range set the value to the lower, or to the higher boundary, respectively.

+*n* Increment the process priority value by *n*.
–*n* Decrement by *n*. This argument can be used only by the privileged user.

nohup [*command*]
Run *command* with HUPs ignored. With no arguments, ignore HUPs throughout the remainder of a script. When given, *command* is always run in a subshell, and the restrictions placed on commands in simple **if** commands apply. All processes detached with **&** are effectively **nohup**'d.

notify [**%***job*] ...
Notify the user asynchronously when the status of the current, or of specified jobs, changes.

onintr [– |*label*]
Control the action of the shell on interrupts. With no arguments, **onintr** restores the default action of the shell on interrupts. (The shell terminates shell scripts and returns to the terminal command input level). With the – argument, the shell ignores all interrupts. With a *label* argument, the shell executes a **goto** *label* when an interrupt is received or a child process terminates because it was interrupted.

popd [**+***n*] Pop the directory stack, and **cd** to the new top directory. The elements of the directory stack are numbered from 0 starting at the top.

+*n* Discard the *n*'th entry in the stack.

pushd [**+***n* |*dir*] Push a directory onto the directory stack. With no arguments, exchange the top two elements.

+*n* Rotate the *n*'th entry to the top of the stack and **cd** to it.
dir Push the current working directory onto the stack and change to *dir*.

rehash Recompute the internal hash table of the contents of directories listed in the *path* variable to account for new commands added.

repeat *count command*
Repeat *command count* times. *command* is subject to the same restrictions as with the one-line **if** statement.

set [*var* [**=** *value*]]
set *var***[***n***] =** *word*
With no arguments, **set** displays the values of all shell variables. Multiword values are displayed as a parenthesized list. With the *var* argument alone, **set** assigns an empty (null) value to the variable *var*. With arguments of the form *var* **=** *value* **set** assigns *value* to *var*, where *value* is one of:

word A single word (or quoted string).
(*wordlist***)** A space-separated list of words enclosed in parentheses.

Values are command and filename expanded before being assigned. The form **set** *var***[***n***] =** *word* replaces the *n*'th word in a multiword value with *word*.

Multiple assignments can be performed with a single **set** command:

```
set notify mail=(30 /usr/mail/nemo)
```

setenv [*VAR* [*word*]]
With no arguments, **setenv** displays all environment variables. With the *VAR* argument sets the environment variable *VAR* to have an empty (null) value. (By convention, environment variables are normally given upper-case names.) With both *VAR* and *word* arguments **setenv** sets the environment variable **NAME** to the value *word*, which must be either a single word or a quoted string. The most commonly used environment variables, **USER**, **TERM**, and **PATH**, are automatically imported to and exported from the **csh** variables **user**, **term**, and **path**; there is no need to use **setenv** for these. In addition, the shell sets the **PWD** environment variable from the **csh** variable **cwd** whenever the latter changes.

shift [*variable*]
The components of **argv**, or *variable*, if supplied, are shifted to the left, discarding the first component. It is an error for the variable not to be set, or to have a null value.

source [–**h**] *name*
Reads commands from *name*. **source** commands can be nested, but if they are nested too deeply the shell may run out of file descriptors. An error in a sourced file at any level terminates all nested **source** commands.

–**h** Place commands from the file *name* on the history list without executing them.

stop [**%***job*] ... Stop the current or specified background job.

suspend Stop the shell in its tracks, much as if it had been sent a stop signal with **^Z**. This is most often used to stop shells started by **su**.

switch (*string***)**
case *label***:**
...
breaksw
...
default:
...
breaksw
endsw Each *label* is successively matched, against the specified *string*, which is first command and filename expanded. The file metacharacters *****, **?** and **[**...**]** can be used in the case labels, which are variable expanded. If none of the labels match before a "default" label is found, execution begins after the default label. Each **case** statement and the **default** statement must appear at the beginning of a line. The command **breaksw** continues execution after the **endsw**. Otherwise control falls through subsequent **case** and **default** statements as with C. If no label matches and there is no default, execution continues after the **endsw**.

time [*command*]
With no argument, print a summary of time used by this C shell and its children. With an optional *command*, execute *command* and print a summary of the time it uses.

umask [*value*] Display the file creation mask. With *value* set the file creation mask. *value* is given in octal, and is XORed with the permissions of 666 for files and 777 for directories to arrive at the permissions for new files. Common values include 002, giving complete access to the group, and read (and directory search) access to others, or 022, giving read (and directory search) but not write permission to the group and others.

unalias *pattern*
Discard aliases that match (filename substitution) *pattern*. All aliases are removed by **unalias ***.

unhash Disable the internal hash table.

unlimit [–**h**] [*resource*]
Remove a limitation on *resource*. If no *resource* is specified, then all *resource* limitations are removed. See the description of the **limit** command for the list of *resource* names.

–**h** Remove corresponding hard limits. Only the privileged user can do this.

unset *pattern* Remove variables whose names match (filename substitution) *pattern*. All variables are removed by '**unset ***'; this has noticeably distasteful side-effects.

unsetenv *variable*
Remove *variable* from the environment. Pattern matching, as with **unset** is not performed.

wait Wait for background jobs to finish (or for an interrupt) before prompting.

while (*expr***)**
...
end While **expr** is true (evaluates to non-zero), repeat commands between the **while** and the matching **end** statement. **break** and **continue** can be used to terminate or continue the loop prematurely. The **while** and **end** must appear alone on their input lines. If the shell's input is a terminal, it prompts for commands with a question-mark until the **end** command is entered and then performs the commands in the loop.

% [*job*] [**&**] Bring the current or indicated *job* to the foreground. With the ampersand, continue running *job* in the background.

@ [*var* **=expr**]
@ [*var* **[***n***] =expr**]
With no arguments, display the values for all shell variables. With arguments, the variable *var*, or the *n*'th word in the value of *var* , to the value that **expr** evaluates to. (If **[***n***]** is supplied, both *var* and its *n*'th component must already exist.)

If the expression contains the characters **>**, **<**, **&** or |, then at least this part of **expr** must be placed within parentheses.

The operators ***=**, **+=**, etc., are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of **expr** that would otherwise be single words.

Special postfix operators, **++** and — increment or decrement *name*, respectively.

Environment Variables and Predefined Shell Variables

Unlike the standard shell, the C shell maintains a distinction between environment variables, which are automatically exported to processes it invokes, and shell variables, which are not. Both types of variables are treated similarly under variable substitution. The shell sets the variables **argv**, **cwd**, **home**, **path**, **prompt**, **shell**, and **status** upon initialization. The shell copies the environment variable **USER** into the shell variable **user**, **TERM** into **term**, and **HOME** into **home**, and copies each back into the respective environment variable whenever the shell variables are reset. **PATH** and **path** are similarly handled. You need only set **path** once in the **.cshrc** or **.login** file. The environment variable **PWD** is set from **cwd** whenever the latter changes. The following shell variables have predefined meanings:

argv Argument list. Contains the list of command-line arguments supplied to the current invocation of the shell. This variable determines the value of the positional parameters **$1**, **$2**, and so on. Note: **argv[0]** does not contain the command name.

cdpath Contains a list of directories to be searched by the **cd**, **chdir**, and **popd** commands, if the directory argument each accepts is not a subdirectory of the current directory.

child The process id of the most recently started background job.

cwd The full pathname of the current directory.

echo Echo commands (after substitutions), just before execution.

fignore A list of filename suffixes to ignore when attempting filename completion. Typically the single word '**.o**'.

filec Enable filename completion, in which case the <Ctrl-d> character <Ctrl-d>) and the ESC character have special significance when typed in at the end of a terminal input line:

EOT Print a list of all filenames that start with the preceding string.
ESC Replace the preceding string with the longest unambiguous extension.

hardpaths
If set, pathnames in the directory stack are resolved to contain no symbolic-link components.

histchars
A two-character string. The first character replaces **!** as the history-substitution character. The second replaces the carat (**^**) for quick substitutions.

history The number of lines saved in the history list. A very large number may use up all of the C shell's memory. If not set, the C shell saves only the most recent command.

home The user's home directory. The filename expansion of **~** refers to the value of this variable.

ignoreeof
If set, the shell ignores EOF from terminals. This protects against accidentally killing a C shell by typing a <Ctrl-d>.

mail A list of files where the C shell checks for mail. If the first word of the value is a number, it specifies a mail checking interval in seconds (default 5 minutes).

nobeep Suppress the bell during command completion when asking the C shell to extend an ambiguous filename.

noclobber
Restrict output redirection so that existing files are not destroyed by accident. **>** redirections can only be made to new files. **>>** redirections can only be made to existing files.

noglob Inhibit filename substitution. This is most useful in shell scripts once filenames (if any) are obtained and no further expansion is desired.

nonomatch
Returns the filename substitution pattern, rather than an error, if the pattern is not matched. Malformed patterns still result in errors.

notify If set, the shell notifies you immediately as jobs are completed, rather than waiting until just before issuing a prompt.

path The list of directories in which to search for commands. **path** is initialized from the environment variable **PATH**, which the C shell updates whenever **path** changes. A null word specifies the current directory. The default search path for normal users is: **(. /usr/sbin /usr/bsd /bin /usr/bin /usr/bin/X11)**. For the privileged user, the default search path is: **(/usr/sbin /usr/bsd /bin /usr/bin /etc /usr/etc /usr/bin/X11)**. If **path** becomes unset, only full pathnames execute. An interactive C shell normally hashes the contents of the directories listed after reading **.cshrc**, and whenever **path** is reset. If new commands are added, use the **rehash** command to update the table.

prompt The string an interactive C shell prompts with. Noninteractive shells leave the **prompt** variable unset. Aliases and other commands in the **.cshrc** file that are only useful interactively, can be placed after the following test: '**if ($?prompt == 0) exit**', to reduce startup time for noninteractive shells. A **!** in the **prompt** string is replaced by the current event number. The default prompt is *hostname***%** for mere mortals, or *hostname***#** for the privileged user.

If the **prompt** string includes the sequence **\@***x*, where *x* is one of the characters listed below, it is replaced by the current time and date in the indicated format.

R time as HH:MM AM/PM, for example, 8:40PM
r time as HH:MM:SS AM/PM, for example, 08:40:25 PM
m month of year – 01 to 12
d day of month – 01 to 31
y last 2 digits of year – 00 to 99
D date as mm/dd/yy
H hour – 00 to 23
M minute – 00 to 59
S second – 00 to 59
T time as HH:MM:SS
j day of year – 001 to 366
w day of week – Sunday = 0
a abbreviated weekday – Sun to Sat
h abbreviated month – Jan to Dec
n insert a newline character
t insert a tab character

savehist
The number of lines from the history list that are saved in **~/.history** when the user logs out. Large values for **savehist** slow down the C shell during startup. To prevent **su** sessions from overwriting the underlying user's history file, the shell only writes in the **~/.history** file if its current effective user id is the same as the owner of the directory specified by the **home** variable.

shell The file in which the C shell resides. This is used in forking shells to interpret files that have execute bits set, but that are not executable by the system.

status The status returned by the most recent command. If that command terminated abnormally, 0200 is added to the status. Built-in commands that fail return exit status 1, all other built-in commands set status to 0.

time Control automatic timing of commands. Can be supplied with one or two values. The first is the reporting threshold in CPU seconds. The second is a string of tags and text indicating which resources to report on. A tag is a percent sign (**%**) followed by a single *upper-case* letter (unrecognized tags print as text):

%D Average amount of unshared data space used in Kilobytes.
%E Elapsed (wallclock) time for the command.

%F Page faults.
%I Number of block input operations.
%K Average amount of unshared stack space used in Kilobytes.
%M Maximum real memory used during execution of the process in Kilobytes.
%O Number of block output operations.
%P Total CPU time — U (user) plus S (system) — as a percentage of E (elapsed) time.
%S Number of seconds of CPU time consumed by the kernel on behalf of the user's process.
%U Number of seconds of CPU time devoted to the user's process.
%W Number of swaps.
%X Average amount of shared memory used in Kilobytes.

The default summary display outputs from the **%U**, **%S**, **%E**, **%P**, **%X**, **%D**, **%I**, **%O**, **%F** and **%W** tags, in that order.

Note that the values for **%D**, **%K**, and **%X** always print as zero since the IRIX kernel does not maintain the **getrusage**(3) counters required to calculate them.

verbose Display each command after history substitution takes place.

FILES

~/.cshrc Read at beginning of execution by each shell.
/etc/cshrc Read by login shells before **.cshrc** at login.
/etc/.login Read by login shells before **.cshrc** and after **/etc/cshrc**.
/etc/csh.cshrc
Read by login shells before **.cshrc** and after **/etc/.login**.
~/.login Read by login shells after **.cshrc** at login.
~/.logout Read by login shells at logout.
~/.history Saved history for use at next login.
/usr/bin/sh Standard shell, for shell scripts not starting with a '**#**'.
/tmp/sh* Temporary file for '**<<**'.
/etc/passwd Source of home directories for '**~***name*'.

SEE ALSO

login(1), sh(1), xwsh(1G), access(2), exec(2), fork(2), pipe(2), a.out(4), ascii(5), environ(5), termio(7).

DIAGNOSTICS

You have stopped jobs.
You attempted to exit the C shell with stopped jobs under job control. An immediate second attempt to exit will succeed, terminating the stopped jobs.

NOTES

Words can be no longer than 1024 characters. The system limits argument lists to 1,048,576 characters. However, the maximum number of arguments to a command for which filename expansion applies is 1706. Command substitutions can expand to no more characters than are allowed in the argument list. To detect looping, the shell restricts the number of **alias** substitutions on a single line to 20.

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (that is, wrong) as the job may have changed directories internally.

Shell built-in functions are not stoppable/restartable. Command sequences of the form *a* **;** *b* **;** *c* are also not handled gracefully when stopping is attempted. If you suspend *b*, the shell never executes *c*. This is especially noticeable if the expansion results from an alias. It can be avoided by placing the sequence in parentheses to force it into a subshell.

Control over terminal output after processes are started is primitive.

Multiline shell procedures should be provided, as they are with the standard (Bourne) shell.

Commands within loops, prompted for by **?**, are not placed in the *history* list.

Control structures should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with |, and to be used with **&** and **;** metasyntax.

It should be possible to use the **:** modifiers on the output of command substitutions. There are two problems with **:** modifier usage on variable substitutions: not all of the modifiers are available, and only one modifier per substitution is allowed.

The **g** (global) flag in history substitutions applies only to the first match in each word, rather than all matches in all words. The the standard text editors consistently do the latter when given the **g** flag in a substitution command.

Quoting conventions are confusing. Overriding the escape character to force variable substitutions within double quotes is counterintuitive and inconsistent with the Bourne shell.

Symbolic links can fool the shell. Setting the **hardpaths** variable alleviates this.

'**set path**' should remove duplicate pathnames from the pathname list. These often occur because a shell script or a **.cshrc** file does something like '**set path=(/usr/local /usr/hosts $path)**' to ensure that the named directories are in the pathname list.

The only way to direct the standard output and standard error separately is by invoking a subshell, as follows:

example% (*command* **>** *outfile***) >&** *errorfile*

Although robust enough for general use, adventures into the esoteric periphery of the C shell may reveal unexpected quirks.

NAME

devnm – device name

SYNOPSIS

/etc/devnm [name] ...

DESCRIPTION

devnm identifies the special file associated with the mounted filesystem 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/ips0d0s2 usr
```

if */usr* is mounted on */dev/dsk/ips0d0s2*.

FILES

/dev/dsk/*
/etc/mtab

SEE ALSO

brc(1M).

NAME

dvhtool – modify and obtain disk volume header information

SYNOPSIS

dvhtool [**–b** [**list**] [*bootfile* [*rootpart* [*swappart*]]]]
[**–v** [**creat** *unix_file dvh_file*]
[**add** *unix_file dvh_file*] [**delete** *dvh_file*]
[**get** *dvh_file unix_file*] [**list**]] [*header_filename*]

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 files kept in the volume header area of the disk for standalone use. The partition table describes the logical device partitions. The device parameters describe the specifics of a particular disk drive.

You must be superuser to use *dvhtool*.

Invoked with no arguments (or just a volume header name, *header_filename*), *dvhtool* allows you to examine and modify the disk volume header on the root drive interactively. The **read** command prompts for the name of the device file for the volume header to be worked on. This can be */dev/rvh* for the header of the root disk or the header name of another disk in the */dev/rdsk* directory. See *vh*(7M). It then reads the volume header from the specified device.

The **vd**, **pt**, and **dp** commands first list their respective portions of the volume header and then prompt for modifications. The **write** command writes the possibly modified volume header to the device.

Note: The use of *dvhtool* for changing partitions and parameters is not recommended. Parameters and partitions should be manipulated with *fx*(1M).

Invoked with arguments, *dvhtool* reads the volume header, performs the specified operations, and then writes the volume header. If no *header_filename* is specified on the command line, */dev/rvh* is used.

The following describes *dvhtool* command-line arguments.

–b Allows you to set the current bootfile, root, and swap partitions. The **list** option displays their current settings.

–v Provides five options for modifying and listing the contents of the volume directory information in the disk volume header: **create**, **add**, **delete**, **get**, and **list**.

creat 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.

add Adds a volume directory entry with the name *dvh_file* and the contents of *unix_file*. Unlike the **creat** option, the **add** options do not overwrite an existing entry.

delete Removes the entry named *dvh_file*, if it exists, from the volume directory.

get Copies the requested file from the volume header to the filesystem.

list Lists the current volume directory contents.

SEE ALSO

fx(1M), vh(7M).

NOTE

Several megabytes of disk space may be required in the */tmp* directory when creating or adding files if the free space in the volume header is fragmented. This also makes *dvhtool* run much slower, because all files must be copied to */tmp*, and then back to the volume header.

NAME

ecc – dump memory ECC log

SYNOPSIS

/usr/etc/ecc [**–c**]

DESCRIPTION

ecc dumps the memory error correction code (ECC) log. This log is produced by the memory subsystem each time a memory error occurs. The types of errors are single data or check bit errors (which are automatically corrected) and double bit errors (which are uncorrectable). Some uncorrectable memory errors cause programs to be killed or the system to crash. Correctable errors, while OK, should be watched. Too many correctable errors in a given memory bank may be an early warning signal of future more serious problems.

This command functions only on systems that have error correcting memory.

The **–c** option causes the log to be cleared. Note that on system crash and subsequent reset the log contents are not cleared and can be read using the prom monitor (see *prom*(1)). Upon system boot, the log is cleared.

The various memory configurations cannot be determined by software, so *ecc* prints out the SIM locations for all possible configurations. It is up to service personnel to determine which is appropriate for a given system.

NOTE

This command is only used on Crimson (system type IP17)

NAME

ed, **red** – text editor

SYNOPSIS

ed [–**s**] [–**p** *string*] [–**x**] [–**C**] [*file*]
red [–**s**] [–**p** *string*] [–**x**] [–**C**] [*file*]

DESCRIPTION

ed is the standard text editor. **red** is a restricted version of **ed**. If the *file* argument is given, **ed** simulates an **e** command (see below) on the named file; that is to say, the file is read into **ed**'s buffer so that it can be edited. Both **ed** and **red** process supplementary code set characters in *file*, and recognize supplementary code set characters in the prompt string given to the –**p** option (see below) according to the locale specified in the **LC_CTYPE** environment variable (see **LANG** in **environ**(5)). In regular expressions, pattern searches are performed on characters, not bytes, as described below.

–**s** Suppresses the printing of byte counts by **e**, **r**, and **w** commands, of diagnostics from **e** and **q** commands, and of the **!** prompt after a **!***shell command*.

–**p** Allows the user to specify a prompt string. The string can contain supplementary code set characters.

–**x** Encryption option; when used, **ed** simulates an **X** command and prompts the user for a key. This key is used to encrypt and decrypt text using the algorithm of **crypt**(1). The **X** command makes an educated guess to determine whether text read in is encrypted or not. The temporary buffer file is encrypted also, using a transformed version of the key typed in for the –**x** option. See **crypt**(1). Also, see the NOTES section at the end of this reference page.

–**C** Encryption option; the same as the –**x** option, except that **ed** simulates a **C** command. The **C** command is like the **X** command, except that all text read in is assumed to have been encrypted.

ed operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a **w** (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

red is a restricted version of **ed**. It allows only editing of files in the current directory. It prohibits executing shell commands via **!***shell command*. Attempts to bypass these restrictions result in an error message (restricted shell).

Both **ed** and **red** support the **fspec**(4) formatting capability. After including a format specification as the first line of *file* and invoking **ed** with your terminal in **stty** –**tabs** or **stty tab3** mode (see **stty**(1)), the specified tab stops are automatically used when scanning *file*. For example, if the first line of a file contained:

```
<:t5,10,15 s72:>
```

tab stops are set at columns 5, 10, and 15, and a maximum line length of 72 is imposed. When you are entering text into the file, this format is not in effect; instead, because of being in **stty** −**tabs** or **stty tab3** mode, tabs are expanded to every eighth column.

Commands to **ed** have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command can appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While **ed** is accepting text, it is said to be in *input mode*. In this mode, no commands are recognized; all input is merely collected. Leave input mode by typing a period (**.**) at the beginning of a line, followed immediately by pressing RETURN.

ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (for example, **s**) to specify portions of a line that are to be substituted. A regular expression specifies a set of character strings. A member of this set of strings is said to be matched by the regular expression. The regular expressions allowed by **ed** are constructed as follows:

The following one-character regular expressions match a single character:

1.1 An ordinary character (not one of those discussed in 1.2 below) is a one-character regular expression that matches itself.

1.2 A backslash (\) followed by any special character is a one-character regular expression that matches the special character itself. The special characters are:

a. **.**, *, **[**, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets (**[]**; see 1.4 below).

b. ^ (caret or circumflex), which is special at the beginning of a regular expression (see 4.1 and 4.3 below), or when it immediately follows the left of a pair of square brackets (**[]**) (see 1.4 below).

c. **$** (dollar sign), which is special at the **end** of a regular expression (see 4.2 below).

d. The character that is special for that specific regular expression, that is used to bound (or delimit) a regular expression. (For example, see how slash (**/**) is used in the **g** command, below.)

1.3 A period (**.**) is a one-character regular expression that matches any character, including supplementary code set characters, except newline.

1.4 A non-empty string of characters enclosed in square brackets (**[]**) is a one-character regular expression that matches one character, including supplementary code set characters, in that string. If, however, the first character of the string is a circumflex (^), the one-character regular expression matches any character, including supplementary code set characters, except newline and the remaining characters in the string. The **^** has this special meaning only if it occurs first in the string. The minus (–) can be used to indicate a range of consecutive characters, including supplementary code set characters; for example, **[0–9]** is equivalent to **[0123456789]**. Characters specifying the range must be from the same code set; when the characters are from different code sets, one of the characters specifying the range is matched. The **–** loses this special meaning if it occurs first (after an initial **^**, if any) or last in the string. The right square bracket (**]**) does not terminate such a string when it is the first character within it (after an initial **^**, if any); for example, **[]a–f]** matches either a right square bracket (**]**) or one of the ASCII letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules can be used to construct regular expressions from one-character regular expressions:

2.1 A one-character regular expression is an regular expression that matches whatever the one-character regular expression matches.

2.2 A one-character regular expression followed by an asterisk (*) is a regular expression that matches zero or more occurrences of the one-character regular expression, which can be a supplementary code set character. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character regular expression followed by **\{***m***\}**, **\{***m*,**\}**, or **\{***m*,*n***\}** is a regular expression that matches a range of occurrences of the one-character regular expression. The values of *m* and *n* must be non-negative integers less than 256; **\{***m***\}** matches exactly *m* occurrences; **\{***m*,**\}** matches at least *m* occurrences; **\{***m*,*n***\}** matches any number of occurrences between *m* and *n* inclusive. Whenever a choice exists, the regular expression matches as many occurrences as possible.

2.4 The concatenation of regular expressions is an regular expression that matches the concatenation of the strings matched by each component of the regular expression.

2.5 A regular expression enclosed between the character sequences **\(** and **\)** defines a sub-expression that matches whatever the unadorned regular expression matches. Inside a sub-expression the anchor characters ((^) and (**$**)) have no special meaning and match their respective literal characters.

2.6 The expression *****n* matches the same string of characters as was matched by an expression enclosed between **\(** and **\)** earlier in the same regular expression. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of **\(** counting from the left. For example, the expression ^**\(.*****\)\1$** matches a line consisting of two repeated appearances of the same string.

A regular expression can be constrained to match words.

3.1 **\<** constrains a regular expression to match the beginning of a string or to follow a character that is not a digit, underscore, or letter. The first character matching the regular expression must be a digit, underscore, or letter.

3.2 **\>** constrains a regular expression to match the end of a string or to precede a character that is not a digit, underscore, or letter.

A regular expression can be constrained to match only an initial segment or final segment of a line (or both).

4.1 A circumflex (**^**) at the beginning of a regular expression constrains that regular expression to match an initial segment of a line.

4.2 A dollar sign (**$**) at the end of an entire regular expression constrains that regular expression to match a final segment of a line.

4.3 The construction **^***regular expression***$** constrains the regular expression to match the entire line.

The null regular expression (for example, **//**) is equivalent to the last regular expression encountered. See also the last paragraph of the DESCRIPTION section below.

To understand addressing in **ed** it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; the exact effect on the current line is discussed under the description of each command. *addresses* are constructed as follows:

1. The character **.** addresses the current line.
2. The character **$** addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. **'***x* addresses the line marked with the mark name character *x*, which must be a lower-case letter (**a**–**z**). Lines are marked with the **k** command described below.
5. A regular expression enclosed by slashes (**/**) addresses the first line found by searching forward from the line following the current line toward the end of the buffer and stopping at the first line containing a string matching the regular expression. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph of the DESCRIPTION section below.
6. A regular expression enclosed in question marks (**?**) addresses the first line found by searching backward from the line preceding the current line toward the beginning of the buffer and stopping at the first line containing a string matching the regular expression. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last

paragraph of the DESCRIPTION section below.

7. An address followed by a plus sign (**+**) or a minus sign (–) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. A shorthand for **.+5** is **.5**.

8. If an address begins with **+** or –, the addition or subtraction is taken with respect to the current line; for example, –**5** is understood to mean **.**–**5**.

9. If an address ends with **+** or –, 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of Rule 8, immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to –.) Moreover, trailing **+** and – characters have a cumulative effect, so –– refers to the current line less 2.

10. For convenience, a comma (**,**) stands for the address pair **1,$**, while a semicolon (**;**) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (**,**). They can also be separated by a semicolon (**;**). In the latter case, the first address is calculated, the current line (.) is set to that value, and then the second address is calculated. This feature can be used to determine the starting line for forward and backward searches (see Rules 5 and 6, above). The second address of any two-address sequence must correspond to a line in the buffer that follows the line corresponding to the first address.

In the following list of **ed** commands, the parentheses shown prior to the command are not part of the address; rather they show the default address(es) for the command.

It is generally illegal for more than one command to appear on a line. However, any command (except **e**, **f**, **r**, or **w**) can be suffixed by **l**, **n**, or **p** in which case the current line is either listed, numbered or printed, respectively, as discussed below under the **l**, **n**, and **p** commands.

(.)a
text
.

The **a**ppend command accepts zero or more lines of text and appends it after the addressed line in the buffer. The current line (**.**) is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the ''appended'' text to be placed at the beginning of the buffer. The maximum number of bytes that can be entered from a terminal is 256 per line (including the newline character).

(.)c
text
.

The **c**hange command deletes the addressed lines from the buffer, then accepts zero or more lines of text that replaces these lines in the buffer. The current line (**.**) is left at the last line input, or, if there were none, at the first line that was not deleted.

C

Same as the **X** command, described later, except that **ed** assumes all text read in for the **e** and **r** commands is encrypted unless a null key is typed in.

(.,.)d

The **d**elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

The **e**dit command deletes the entire contents of the buffer and then reads the contents of *file* into the buffer. The current line (**.**) is set to the last line of the buffer. If *file* is not given, the currently remembered filename, if any, is used (see the **f** command). The number of characters read in is printed; *file* is remembered for possible use as a default filename in subsequent **e**, **r**, and **w** commands. If *file* is replaced by **!**, the rest of the line is taken to be a shell (**sh**(1)) command whose output is to be read in. Such a shell command is not remembered as the current filename. See also DIAGNOSTICS below. If *file* is replaced by **%**, and if additional *file* arguments were specified on the command line, the next filename specified on the command line is used.

E *file*

The **E**dit command is like **e**, except that the editor does not check to see if any changes have been made to the buffer since the last **w** command.

f *file*

If *file* is given, the **f**ile-name command changes the currently remembered filename to *file*; otherwise, it prints the currently remembered filename.

(1,$)g/*regular expression***/***command list*

In the **g**lobal command, the first step is to mark every line that matches the given regular expression. Then, for every such line, the given *command list* is executed with the current line (**.**) initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; **a**, **i**, and **c** commands and associated input are permitted. The **.** terminating input mode can be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the **p** command. The **g**, **G**, **v**, and **V** commands are not permitted in the *command list*. See also the NOTES section and the last paragraph of the DESCRIPTION section below.

(1,$)G/*regular expression***/**

In the interactive **G**lobal command, the first step is to mark every line that matches the given regular expression. Then, for every such line, that line is printed, the current line (**.**) is changed to that line, and any one command (other than one of the **a**, **c**, **i**, **g**, **G**, **v**, and **V** commands) can be input and is executed. After the execution of that command, the next marked line is printed, and so on; a newline acts as a null command; an **&** causes the re-execution of the most recent command executed within the current invocation of **G**. Note that the commands input as part of the execution of the **G** command can address and affect any lines in the buffer. The **G** command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

The **h**elp command gives a short error message that explains the reason for the most recent **?** diagnostic.

H

The **H**elp command causes **ed** to enter a mode in which error messages are printed for all subsequent **?** diagnostics. It also explains the previous **?** if there was one. The **H** command alternately turns this mode on and off; it is initially off.

(.)i
text
.

The **i**nsert command accepts zero or more lines of text and inserts it before the addressed line in the buffer. The current line (**.**) is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the **a** command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that can be entered from a terminal is 256 per line (including the newline character).

(.,.+1)j

The **j**oin command joins contiguous lines by removing the appropriate newline characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mar**k** command marks the addressed line with name *x*, which must be a lower-case letter (**a**–**z**). The address ′*x* then addresses this line; the current line (**.**) is unchanged.

(.,.)l

The **l**ist command prints the addressed lines in an unambiguous way: a few non-printing characters (for example, tab, backspace) are represented by visually mnemonic overstrikes. All other non-printing characters are printed in octal, and long lines are folded. An **l** command can be appended to any command other than **e**, **f**, **r**, or **w**.

(.,.)m*a*

The **m**ove command repositions the addressed line(s) after the line addressed by *a*. Address **0** is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address *a* falls within the range of moved lines; the current line (**.**) is left at the last line moved.

(.,.)n

The **n**umber command prints the addressed lines, preceding each line by its line number and a tab character; the current line (**.**) is left at the last line printed. The **n** command can be appended to any command other than **e**, **f**, **r**, or **w**.

(.,.)p

The **p**rint command prints the addressed lines; the current line (**.**) is left at the last line printed. The **p** command can be appended to any command other than **e**, **f**, **r**, or **w**. For example, **dp** deletes the current line and prints the new current line.

P

The editor prompts with a * for all subsequent commands. The **P** command alternately turns this mode on and off; it is initially off.

q

The **q**uit command causes **ed** to exit. No automatic write of a file is done; however, see DIAGNOSTICS below.

Q

The editor exits without checking if changes have been made in the buffer since the last **w** command.

($)r *file*

The **r**ead command reads the contents of *file* into the buffer. If *file* is not given, the currently remembered filename, if any, is used (see the **e** and **f** commands). The currently remembered filename is not changed unless *file* is the very first filename mentioned since **ed** was invoked. Address 0 is legal for **r** and causes the file to be read in at the beginning of the buffer. If the read is successful, the number of characters read in is printed; the current line (**.**) is set to the last line read in. If *file* is replaced by **!**, the rest of the line is taken to be a shell (see **sh**(1)) command whose output is to be read in. For example, **$r !ls** appends current directory to the end of the file being edited. Such a shell command is not remembered as the current filename.

(.,.)s/*regular expression***/***replacement***/** or
(.,.)s/*regular expression***/***replacement***/g** or
(.,.)s/*regular expression***/***replacement***/***n* *n* = 1-512

The **s**ubstitute command searches each addressed line for an occurrence of the specified regular expression. In each line in which a match is found, all (non-overlapped) matched strings are replaced by the *replacement* if the global replacement indicator **g** appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. If a

number *n*, appears after the command, only the *n*-th occurrence of the matched string on each addressed line is replaced. It is an error if the substitution fails on all addressed lines. Any character other than space or newline can be used instead of **/** to delimit the regular expression and the *replacement*; the current line (**.**) is left at the last line on which a substitution occurred. See also the last paragraph of the DESCRIPTION section below.

An ampersand (**&**) appearing in the *replacement* is replaced by the string matching the regular expression on the current line. The special meaning of **&** in this context can be suppressed by preceding it by ****. As a more general feature, the characters *****n*, where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression of the specified regular expression enclosed between **\(** and **\)**. When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of **\(** starting from the left. When the character **%** is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The **%** loses its special meaning when it is in a replacement string of more than one character or is preceded by a ****.

A line can be split by substituting a newline character into it. The newline in the *replacement* must be escaped by preceding it by ****. Such substitution cannot be done as part of a **g** or **v** command list.

(.,.)t*a*
This command acts just like the **m** command, except that a copy of the addressed lines is placed after address **a** (which can be 0); the current line (**.**) is left at the last line copied.

u
The **u**ndo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent **a**, **c**, **d**, **g**, **i**, **j**, **m**, **r**, **s**, **t**, **v**, **G**, or **V** command.

(1,$)v/*regular expression***/***command list*
This command is the same as the global command **g**, except that the lines marked during the first step are those that do not match the regular expression.

(1,$)V/*regular expression***/**
This command is the same as the interactive global command **G**, except that the lines that are marked during the first step are those that do not match the regular expression.

(1,$)w *file*
The **w**rite command writes the addressed lines into *file*. If *file* does not exist, it is created with mode **666** (readable and writable by everyone), unless your file creation mask dictates otherwise; see the description of the **umask** special command on **sh**(1). The currently remembered filename is not changed unless *file* is the very first filename mentioned since **ed** was invoked. If no filename is given, the currently remembered filename, if any, is used (see the **e** and **f** commands); the current line (**.**) is unchanged. If the command is successful, the number of characters written is printed. If *file* is replaced by **!**, the rest of the line is taken to be a shell (see **sh**(1)) command whose standard input is the addressed lines. Such a shell command is not remembered as the current filename.

(1,$)W *file*
This command is the same as the **w**rite command above, except that it appends the addressed lines to the end of *file* if it exists. If *file* does not exist, it is created as described above for the **w** command.

X
A key is prompted for, and it is used in subsequent **e**, **r**, and **w** commands to decrypt and encrypt text using the **crypt**(1) algorithm. An educated guess is made to determine whether text read in for the **e** and **r** commands is encrypted. A null key turns off encryption. Subsequent **e**, **r**, and **w** commands use this key to encrypt or decrypt the text (see **crypt**(1)). An explicitly empty key turns off encryption. Also, see the **–x** option of **ed**.

($)=
The line number of the addressed line is typed; the current line (**.**) is unchanged by this command.

!*shell command*
The remainder of the line after the **!** is sent to the UNIX system shell (see **sh**(1)) to be interpreted as a command. Within the text of that command, the unescaped character **%** is replaced with the remembered filename; if a **!** appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, **!!** repeats the last shell command. If any expansion is performed, the expanded line is echoed; the current line (**.**) is unchanged.

(.+1)<newline>
An address alone on a line causes the addressed line to be printed. A newline alone is equivalent to **.+1p**; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII DEL or BREAK) is sent, **ed** prints a **?** and returns to its command level.

Some size limitations: 512 bytes in a line, 256 bytes in a global command list, and 1024 bytes in the pathname of a file (counting slashes). The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

When reading a file, **ed** discards ASCII NUL characters.

If a file is not terminated by a newline character, **ed** adds one and puts out a message explaining what it did.

If the closing delimiter of a regular expression or of a replacement string (for example, **/**) would be the last character before a newline, that delimiter can be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

```
s/s1/s2     s/s1/s2/p
g/s1        g/s1/p
?s1         ?s1?
```

FILES

$TMPDIR if this environmental variable is not null, its value is used in place of **/var/tmp** as the directory name for the temporary work file

/var/tmp if **/var/tmp** exists, it is used as the directory name for the temporary work file

/tmp if the environmental variable **TMPDIR** does not exist or is null, and if **/var/tmp** does not exist, **/tmp** is used as the directory name for the temporary work file

ed.hup work is saved here if the terminal is hung up

/usr/lib/locale/*locale***/LC_MESSAGES/uxcore.abi**
language-specific message file (see **LANG** in **environ** (5))

SEE ALSO

edit(1), ex(1), grep(1), sed(1), sh(1), stty(1), umask(1), vi(1), fspec(4), regexp(5).

DIAGNOSTICS

? Command errors. Type the **h** command for a short error message.

?*file* An inaccessible file. (Use the **h**elp and **H**elp commands for detailed explanations.)

If changes have been made in the buffer since the last **w** command that wrote the entire buffer, **ed** warns the user if an attempt is made to destroy **ed**'s buffer via the **e** or **q** commands. It prints **?** and allows one to continue editing. A second **e** or **q** command at this point takes effect. The **−s** command-line option inhibits this feature.

NOTES

The − option, although it continues to be supported, has been replaced in the documentation by the **−s** option that follows the Command Syntax Standard (see **intro**(1)).

A **!** command cannot be subject to a **g** or a **v** command.

The **!** command and the **!** escape from the **e**, **r**, and **w** commands cannot be used if the editor is invoked from a restricted shell (see **sh**(1)).

The sequence **\n** in a regular expression does not match a newline character.

If the editor input is coming from a command file (for example, **ed** *file* **<** *ed_cmd_file*), the editor exits at the first failure.

NAME

find – find files

SYNOPSIS

find path-name-list [expression]

DESCRIPTION

find recursively descends the directory hierarchy for each pathname in the *path-name-list* (that is, one or more pathnames) seeking files that match a boolean *expression* written in the primaries given below. If the expression does not contain at least one of **−print**, **−ok**, or **−exec**, including the case of a null expression, a **−print** is implicit. In the descriptions, the argument *n* is used as a decimal integer where **+***n* means more than *n*, **−***n* means less than *n*, and *n* means exactly *n*. Valid expressions are:

−name *file* — True if *file* matches the current filename. Normal shell argument syntax can be used if escaped (watch out for **[**, **?**, and **∗**).

−perm [**−**]*mode* — True if the file permission flags exactly match the file mode given by *mode* which can be an octal number or a symbolic expression of the form used in *chmod*(1)). If *mode* is prefixed by a minus sign, only the bits that are set in *mode* are compared with the file permission flags, and the expression evaluates true if they match.

−type *c* — True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **l**, **p**, **f**, or **s** for block special file, character special file, directory, symbolic link, fifo (a.k.a named pipe), plain file, or socket respectively.

−links *n* — True if the file has *n* links.

−user *uname* — True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the */etc/passwd* file, it is taken as a user ID.

−nouser — True if the file belongs to a user not in the */etc/passwd* file.

−group *gname* — True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the */etc/group* file, it is taken as a group ID.

−nogroup — True if the file belongs to a group not in the */etc/group* file.

−size *n*[**c**] — True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a **c**, the size is in characters.

−inum *n* — True if *n* is the inode number of the file.

−atime *[+−]n* — True if the file was accessed *n* days ago. The definition of *n* days ago is any time within the interval beginning exactly *n**24 hours ago and ending exactly (*n*−1)*24 hours ago. The **+** and − prefixes signify more or less than *n* days ago, respectively, thus **+***n* means more than *n**24 hours ago, and **−***n* means less than *n**24 hours ago. (See *stat*(2) for a

description of which file operations change the access time of a file.) The access time of directories in *path-name-list* is changed by *find* itself.

−mtime *[+−]n* True if the file was modified *n* days ago. See **−atime** for definition of "*n* days ago". (See *stat*(2) for a description of which file operations change the modification time of a file.)

−ctime *[+−]n* True if the file was changed *n* days ago. See **−atime** for definition of "*n* days ago". (See *stat*(2) for a description of which file operations change the change time of a file.)

−exec *cmd* True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument **{}** is replaced by the current pathname.

−ok *cmd* Like **−exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.

−print Always true; causes the current pathname to be printed.

−cpio *device* Always true; write the current file on *device* in *cpio*(1) format (5120-byte records). **find −cpio** issues a warning if it encounters a file larger than two gigabytes. *cpio*(1) must be used to archive files of this size.

−newer *file* True if the current file has been modified more recently than the argument *file* (see *stat*(2) for a description of which file operations change the modification time of a file).

−anewer *file* True if current file has been accessed more recently than the argument *file* (see *stat*(2) for a description of which file operations change the access time of a file).

−cnewer *file* True if current file has been changed more recently than the argument *file* (see *stat*(2) for a description of which file operations change the change time of a file).

−depth Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*(1) to transfer files that are contained in directories without write permission.

−prune Always true; do not examine any directories or files in the directory structure below the *pattern* just matched. If the current pathname is a directory, *find* does not descend into that directory, provided **−depth** is not also used.

−mount Always true; restricts the search to the filesystem containing the current element of the *path-name-list*.

−fstype *type* True if the filesystem to which the file belongs is of type *type*.

–local True if the file physically resides on the local system; causes the search not to descend into remotely mounted filesystems.

–follow Always true; causes the underlying file of a symbolic link to be checked rather than the symbolic link itself.

\(*expression* **\)** True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries can be combined using the following operators (in order of decreasing precedence):

- The negation of a primary (**!** is the unary **not** operator).
- Concatenation of primaries (the **and** operation is implied by the juxtaposition of two primaries).
- Alternation of primaries (**–o** is the **or** operator).

EXAMPLES

To remove all files named *a.out* or **.o* that have not been accessed for a week:

```
find / \( –name a.out –o –name '*.o' \) –atime +7 –exec rm {} \;
```

To display all character special devices on the root filesystem except those under any *dev* directory:

```
find / –mount \( –type d –name dev –prune \) –o –type c –print
```

FILES

/etc/passwd UID information supplier
/etc/group GID information supplier

SEE ALSO

chmod(1), cpio(1), sh(1), test(1), stat(2), umask(2), efs(4), xfs(4).

BUGS

find / –depth always fails with the message:

```
find:  stat failed:  : No such file or directory
```

find relies on a completely correct directory hierarchy for its search. In particular, if a directory's '..' is missing or incorrect, *find* fails at that point and issue some number of these messages:

```
stat failed:
```

–depth and **–prune** do not work together well.

NAME

fsck – check and repair filesystems for EFS

SYNOPSIS

/etc/fsck [**–c**] [**–f**] [**–g**] [**–m**] [**–n**] [**–q**] [**–y**] [**–l** dir] [filesystems]

DESCRIPTION

fsck is applicable only to EFS filesystems.

fsck audits and repairs inconsistent conditions for filesystems. You must have both read and write permission for the device containing the filesystem unless you give the **–n** flag, in which case only read permission is required.

If the filesystem is inconsistent, you are normally prompted for concurrence before each correction is attempted. Most corrective actions result in some loss of data. The amount and severity of data loss can be determined from the diagnostic output. The default action for each correction is to wait for you to respond **yes** or **no**. However, certain option flags cause *fsck* to run in a non-interactive mode.

On completion, the number of files, blocks used, and blocks free are reported.

Note: Checking the raw device is almost always faster.

The following options are accepted by *fsck*:

–c Checks the filesystem only if the superblock indicates that it is dirty, otherwise a message is printed saying that the filesystem is clean and no check is performed. The default in the absence of this option is to always perform the check.

–f Fast check. Check block and sizes and check the free list. The free list is reconstructed if it is necessary. No directory or pathname checks are performed.

–g A low risk "gentle" mode, similar to BSD preen. Problems that do not present any risk of data loss are fixed: these include bad link counts, bad free list, and dirty superblock. If any serious damage is encountered that cannot be repaired without risk of data loss, *fsck* terminates with a warning message.

–l Allows a directory on a mounted filesystem, located elsewhere on the system, to be specified as a salvage directory. Unreferenced regular files, named after their inode numbers, are copied into this salvage directory. This allows files to be salvaged from very badly corrupted filesystems that may not be repairable in place -- if the root inode is lost, for example.

–m Forks multiple instances of *fsck* to check filesystems in parallel for improved speed. This option is effective only when *fsck* is working from the filesystems listed in */etc/fstab* and is ignored if explicit filesystem arguments are given. Also, when this option is specified, entries in */etc/fstab* with the **noauto** option are ignored.

−n Assumes a **no** response to all questions asked by *fsck*; does not open the filesystem for writing.

−q Quiet *fsck*. This option is effectively a version of the **−y** option with less verbose output.

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

If no *filesystems* are specified, *fsck* reads a list of default filesystems from the file */etc/fstab*. This does not include the root filesystem; *fsck* runs on root only if this is explicitly specified.

Normally, a filesystem must be unmounted in order to run *fsck* on it, an error message is printed and no action taken if invoked on a mounted filesystem. The one exception to this is the root filesystem, which must be mounted to run *fsck*. If inconsistencies are detected when running on root, *fsck* causes a remount of root.

PARALLEL OPERATION

When invoked with the **−m** flag and without explicit filesystem parameters, *fsck* scans */etc/fstab* and attempts to fork a check process for each **efs** filesystem found. These checks proceed in parallel, for improved speed.

The name of the device holding the filesystem is printed as each check begins. However, to avoid confusion, the remaining output from these parallel checks is not printed; instead it is placed in log files in the directory */etc/fscklogs*. This directory is created if it does not currently exist.

The log files are named after the last component of the pathname of the device where the filesystem resides. For example, if a filesystem was on */dev/dsk/ips0d1s7* the logfile is named */etc/fscklogs/ips0d1s7*.

Because there is no interaction with the checks, the **−m** option is accepted only in combination with another option implying non interactive behavior: **−y** or **−g**.

As each check completes, the name of the device is printed along with a message indicating success or failure. In the event of failure, the name of the logfile containing the output from the check of that filesystem is also printed.

Some control over the parallelization is possible by placing *passnumbers* in */etc/fstab* (see *fstab*(4)). If pass numbers are given for filesystems, they are checked in the order of their pass numbers. All filesystems with a given pass number are checked (in parallel, if more than one filesystem has the same pass number) before the next highest pass number. A missing pass number defaults to zero. If no pass numbers are present, all filesystems are checked simultaneously if possible.

Note: In fact, *fsck* takes note of the amount of memory available in the system, and limits the number of simultaneous check processes to avoid swapping. If there is not enough memory to avoid swapping for a particular filesystem, the message

```
Warning - Low free memory, swapping likely
```

is printed. If this occurs when *fsck*'ing the root or usr filesystem after a crash, the crash dump is lost. In this case the *fsck* takes longer, but the results are otherwise normal.

CHECKS PERFORMED

Inconsistencies checked are as follows:

1. Inode block addressing checks: Too many direct or indirect extents, extents out of order, bad magic number in extents, blocks that are not in a legal data area of the filesystem, blocks that are claimed by more than one inode.

2. Size checks: Number of blocks claimed by inode inconsistent with inode size, directory size not block aligned.

3. Directory checks: Illegal number of entries in a directory block, bad freespace pointer in directory block, entry pointing to unallocated or outrange inode, overlapping entries, missing or incorrect dot and dotdot entries.

4. Pathname checks: Files or directories not referenced by a pathname starting from the filesystem root.

5. Link count checks: Link counts that do not agree with the number of directory references to the inode.

6. Freemap checks: Blocks claimed free by the freemap but also claimed by an inode, blocks unclaimed by any inode but not appearing in the freemap.

7. Super Block checks: Total free block and/or free i-node count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the your concurrence, reconnected by placing them in the *lost+found* directory, if the files are nonempty. You are 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* forces the reconnection of nonempty directories. The name assigned is the i-node number. The directory *lost+found* must preexist in the root of the filesystem being checked and must have empty slots in which entries can be made. This directory is always created by *mkfs*(1M) when a filesystem is first created.

SUPERBLOCKS AND FILESYSTEM ROBUSTNESS

In IRIX 3.3 and later, a replicated superblock exists in the EFS filesystem, situated at the end of the filesystem space. If *fsck* cannot read the primary superblock it attempts to use the replicated superblock. It prints a message to notify you of the situation. This is automatic; no user intervention is required. Further, *fsck* attempts to determine if a replicated superblock exists, and if not, optionally creates one. Thus, older filesystems benefit from this feature.

Finally, if no superblock can be found on a damaged filesystem, it may be possible to regenerate one by using the new **–r** option of *mkfs*(1M), and then use *fsck* to salvage the filesystem.

Warning: This is **not** effective if the filesystem was created under a version of IRIX other than the currently running version, since *mkfs* defaults have changed from release to release.

OBSOLETE OPTIONS

The options **–b**, **–D**, **–s**, **–S**, and **-t**, which were supported by earlier versions of *fsck*, are now obsolete.

The **–b** option caused a reboot of the system when *fsck* was run on the root filesystem and errors were detected. The behavior now is always to remount the root filesystem in this case.

The **–t** option specified a scratch file for temporary storage; this is now never required.

The **–D** option added extra directory checks; these are now always done by default.

The **–s** and **–S** options caused conditional or forced rebuild of the freelist. The freelist is now exhaustively checked and is always rebuilt if necessary.

All of these options are now legal no-ops.

FILES

/etc/fstab default list of filesystems to check

SEE ALSO

findblk(1M), fpck(1M), mkfs(1M), ncheck(1M), uadmin(2), filesystems(4), fstab(4).

NAME

fsdb – filesystem debugger for EFS

SYNOPSIS

fsdb [**–?**] [**–o**] [**–p'***string***'**] [**–w**] *special*

DESCRIPTION

fsdb is applicable only to EFS filesystems.

fsdb can be used to patch up a damaged filesystem 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 inode. These greatly simplify the process of correcting control block entries or descending the filesystem tree.

Because *fsdb* reads the disk raw, it is able to circumvent normal filesystem security. It also bypasses the buffer cache mechanism. Hence, it is not advisable to use *fsdb* to write to a mounted filesystem.

The options available to *fsdb* are:

–? Display usage.

–o Override some error conditions.

–p'*string***'** Set prompt to *string*.

–w Open for write.

fsdb contains several error-checking routines to verify inode and block addresses. These can be disabled if necessary by invoking *fsdb* with the **–o** option or by the use of the **o** command.

special is the name of a character device file. *fsdb* searches */etc/fstab* for the raw character device filename, if given the name of a filesystem. 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. Since *fsdb* opens the raw device file, any write-throughs bypass the filesystem buffer cache, resulting in a potential mismatch between on-disk and buffer cache data structures. Hence, it is recommended that *fsdb* not be used to write to a mounted filesystem. Note that in order to modify any portion of the disk, *fsdb* must be invoked with the **–w** option.

Wherever possible, *adb*–like syntax was adopted to promote the use of *fsdb* through familiarity.

Numbers are considered hexadecimal by default. However, the user has control over how data are to be displayed or accepted. The *base* command displays or sets the input/output base. Once set, all input defaults to this base and all output is shown in this base. The base can be overridden temporarily for input by preceding hexadecimal numbers with **0x**, decimal numbers with **0t**, or octal numbers with **0**. Hexadecimal numbers beginning with **a–f** or **A–F** must be preceded with **0x** to distinguish them from commands.

Disk addressing by *fsdb* is at the byte level. However, *fsdb* offers many commands to convert a desired inode, directory entry, block, superblock, and so forth to a byte address. Once the address has been calculated, *fsdb* records the result in *dot*.

Several global values are maintained by *fsdb*: the current base (referred to as *base*), the current address (referred to as *dot*), the current inode (referred to as *inode*), the current count (referred to as *count*), and the current type (referred to as *type*). Most commands use the preset value of *dot* in their execution. For example,

```
> 2:inode
```

first sets the value of *dot* to 2, **:** alerts the start of a command, and the *inode* command sets *inode* to 2. A count is specified after a **,**. Once set, *count* remains at this value until a new command is encountered, which resets the value back to 1 (the default). So, if

```
> 2000,400/X
```

is typed, 400 hexadecimal longs are listed from 2000, and when completed the value of *dot* is 2000 + 400 * sizeof (long). If a carriage return is then typed, the output routine uses the current values of *dot*, *count*, and *type* and displays 400 more hexadecimal longs. An ***** causes the entire block to be displayed.

End of block and file are maintained by *fsdb*. When displaying data as blocks, an error message is displayed when the end of the block is reached. When displaying data using the *db*, *directory*, or *file* commands, an error message is displayed if the end of file is reached. This is needed primarily to avoid passing the end of a directory or file and getting unknown and unwanted results.

Examples showing several commands and the use of carriage return are:

```
> 2:ino; 0:dir?d
```

or

```
> 2:ino; 0:db:block?d
```

The two examples are synonymous for getting to the first directory entry of the root of the filesystem. Once there, subsequent carriage returns (or + or –) advance to subsequent entries.

Note that:

```
> 2:inode; :ls /
```

or

```
> 2:inode
> :ls /
```

is again synonymous.

EXPRESSIONS

fsdb recognizes the following symbols. There should be no white space between the symbols and the arguments.

carriage return
Update the value of *dot* by the current value of *type* and display using the current value of *count*.

Numeric expressions can be composed of +, –, *, and % operators (evaluated left to right) and can use parentheses. Once evaluated, the value of *dot* is updated.

,*count* Count indicator. The global value of *count* is updated to *count*. The value of *count* remains until a new command is run. A count specifier of * attempts to show a *block*'s worth of information. The default for *count* is 1.

?*f* Display in structured style with format specifier *f* (see FORMATTED OUTPUT section).

/*f* Display in unstructured style with format specifier *f* (see FORMATTED OUTPUT section).

. The value of *dot*.

+*e* Increment the value of *dot* by the expression *e*. The amount actually incremented is dependent on the size of *type*:

dot = dot + e * sizeof (type)

The default for *e* is 1.

–*e* Decrement the value of *dot* by the expression *e* (see +).

*****e* Multiply the value of *dot* by the expression *e*. Multiplication and division do not use *type*. In the above calculation of *dot*, consider the sizeof (*type*) to be 1.

%*e* Divide the value of *dot* by the expression *e* (see *).

<*name* Restore an address saved in register *name*. *name* must be a single letter or digit.

>*name* Save an address in register *name*. *name* must be a single letter or digit.

=*f* Display indicator. If *f* is a legitimate format specifier (see FORMATTED OUTPUT section), then the value of *dot* is displayed using format specifier *f*. Otherwise, assignment is assumed (see next item).

=[*e*]

=[*s*] Assignment indicator. The address pointed to by *dot* has its contents changed to the value of the expression *e* or to the ASCII representation of the quoted (") string *s*. This may be useful for changing directory names or ASCII file information.

=+*e* Incremental assignment. The address pointed to by *dot* has its contents incremented by expression *e*.

=-*e* Decremental assignment. The address pointed to by *dot* has its contents decremented by expression *e*.

COMMANDS

A command must be prefixed by a **:** character. Only enough letters of the command to uniquely distinguish it are needed. Multiple commands can be entered on one line by separating them by a space, tab, or **;**.

In order to view a potentially unmounted disk in a reasonable manner, *fsdb* offers the *cd*, *pwd*, *ls*, and *find* commands. The functionality of these commands substantially matches those of its IRIX counterparts (see individual commands for details). The *****, **?**, and **[–]** wildcard characters are available.

base=*b* Display or set base. As stated above, all input and output is governed by the current *base*. If the **=b** is left off, the current *base* is displayed. Otherwise, the current *base* is set to *b*. Note that this is interpreted using the old value of *base*. To ensure correctness, use the **0**, **0t**, or **0x** prefix when changing the *base*. The default for *base* is hexadecimal.

block Convert the value of *dot* to a block address.

cd *dir* Change the current directory to directory *dir*. The current values of *inode* and *dot* are also updated. If no *dir* is specified, then change directories to inode 2 (*/*).

cg Convert the value of *dot* to a cylinder group.

directory If the current *inode* is a directory, then the value of *dot* is converted to a directory slot offset in that directory. *dot* now points to this entry.

file The value of *dot* is taken as a relative block count from the beginning of the file. The value of *dot* is updated to the first byte of this block.

find *dir* [**−name** *n*] [**−inum** *i*]
Find files by name or i-number. *find* recursively searches directory *dir* and below for filenames whose i-number matches *i* or whose name matches pattern *n*. Note that only one of the two options (**−name** or **−inum**) can be used at one time. Also, the **−print** is not needed or accepted.

fill=*p* Fill an area of disk with pattern *p*. The area of disk is delimited by *dot* and *count*.

inode Convert the value of *dot* to an inode address. If successful, the current value of *inode* is updated as well as the value of *dot*. As a convenient shorthand, if **:inode** appears at the beginning of the line, the value of *dot* is set to the current *inode* and that inode is displayed in inode format.

ls [**−R**] [**−l**] *pat1 pat2 ...*
List directories or files. If no file is specified, the current directory is assumed. Either or both of the options can be used (but, if used, must be specified before the filename specifiers). Also, as stated above, wildcard characters are available and multiple arguments can be given. The long listing shows only the i-number and the name; use the *inode* command with **?i** to get more information. The output is sorted in alphabetical order. If either the **−R** or the **−l** options is used, then the files can have a character following the filename, indicating the type of the file. Directories have a **/**, symbolic links have a **@**, AF_UNIX address family sockets have a **=** and fifos have an **f**. Regular files and block and character device files have an ***** if they are executable. If the file type is unknown, then a **?** is printed.

override Toggle the value of override. Some error conditions can be overridden if override is toggled on.

prompt *p* Change the *fsdb* prompt to *p*. *p* must be surrounded by double quotes (").

pwd Display the current working directory.

quit Quit *fsdb*.

sb The value of *dot* is taken as the basic block number and then converted to the address of the superblock in that cylinder group. As a shorthand, **:sb** at the beginning of a line sets the value of *dot* to the superblock and displays it in superblock format.

! sh Escape to shell.

INODE COMMANDS

In addition to the above commands, there are several commands that deal with inode fields and operate directly on the current *inode* (they still require the **:**). They can be used to display more easily or change the particular fields. The value of *dot* is only used by the **:db**, **:len**, and **:off** commands. Upon completion of the command, the value of *dot* is changed to point to that particular field. For example,

```
> :ln=+1
```

increments the link count of the current *inode* and set the value of *dot* to the address of the link count field. It is important to know the format of the disk inode structure and the size and alignment of the respective fields; otherwise the output of these commands is not coherent. The disk inode structure is available in <*sys/fs/efs_ino.h*>.

at Access time.

ct Creation time.

db Use the current value of *dot* as an index into the list of extents stored in the disk inode to get the starting disk block number associated with the corresponding extent. Extents number from 0 to 11. In order to display the block itself, you need to pipe this result into the *block* command. For example,

> 1:db:block,20/X

gets the contents of disk block number field of extent number 1 from the inode and converts it to a block address. Twenty longs are then displayed in hexadecimal (see the FORMATTED OUTPUT section).

gen Inode generation number.

gid Group ID.

ln Link count.

len Use the current value of *dot* as an index into the list of extents stored in the disk inode to get the length associated with the corresponding extent. Extents number from 0 to 11. This field is one byte long. For example,

> 1:len/b

displays the contents of the len field of extent number 1.

mt Modification time.

md Mode.

maj Major device number.

min Minor device number.

nex Number of extents.

nm Although listed here, this command actually operates on the directory name field. Once poised at the desired directory entry (using the *directory* command), this command allows you to change or display the directory name. For example,

```
> 7:dir:nm="foo"
```

gets the seventh directory entry of the current *inode* and changes its name to foo. Names have to be the same size as the original name. If the new name is smaller, it is padded with **#**. If it is larger, the string is truncated to fit and a warning message to this effect is displayed.

off Use the current value of *dot* as an index into the list of extents stored in the disk inode to get the logical block offset associated with the corresponding extent. Extents number from 0 to 11. This field is three bytes long. For example,

```
> 3:off,3/b
```

displays the contents of the off field of extent number 3.

sz File size.

uid User ID.

FORMATTED OUTPUT

There are two styles and many format types. The two styles are structured and unstructured. Structured output is used to display inodes, directories, superblocks, and the like. Unstructured output only displays raw data. The following table shows the different ways of displaying:

?

- **c** Display as cylinder groups
- **i** Display as inodes
- **I** Display as inodes (all direct extents)
- **d** Display as directories
- **s** Display as superblocks
- **e** Display as extents

/

b Display as bytes
c Display as characters
o O Display as octal shorts or longs
d D Display as decimal shorts or longs
x X Display as hexadecimal shorts or longs

The format specifier immediately follows the **/** or **?** character. The values displayed by **/b** and all **?** formats are displayed in the current *base*. Also, *type* is appropriately updated upon completion.

EXAMPLES

> :base — Display the current input/output base (hexadecimal by default).

> :base=0xa — Change the current input/output base to decimal.

> 0t2000+(0t400%(0t20+0t20))=D
Display 2010 in decimal (use of *fsdb* as a calculator for complex arithmetic). The **0t** indicates that the numbers are to be interpreted as decimal numbers and are necessary only if the current base is not decimal. Brackets should be used to force ordering since *fsdb* does not force the normal ordering of operators. Note that **%** is the division symbol.

> 386:ino?i — Display i-number 386 in an inode format. This now becomes the current *inode*.

> :ln=4 — Change the link count for the current *inode* to 4.

> :ln/x — Display the link count as a hexadecimal short.

> :ln=+1 — Increment the link count by 1.

> :sz/D — Display the size field as a decimal long.

> :sz/X — Display the size field as a hexadecimal long.

> :ct=X — Display the creation time as a hexadecimal long.

> :mt=t — Display the modification time in time format.

> 0:db,3/b — Display the block number of the first extent as 3 bytes. The block number has to be printed out as bytes, because of alignment considerations.

> 0:file/c — Display, in ASCII, block zero of the file associated with the current *inode*.

> 5:dir:inode; 0:file,*/c
Change the current inode to that associated with the fifth directory entry (numbered from zero) of the current *inode*. The first logical block of the file is then displayed in ASCII.

> :sb Display the superblock of this filesystem.

> 0:cg?c Display cylinder group information and summary for the first cylinder group (cg number 0).

> 7:dir:nm="name"
Change the name field in the directory slot to *name*.

> 2:db:block,*?d Display the third block of the current *inode* as directory entries.

> 0:db=0x43b Change the disk block number associated with extent 0 of the inode to 0x43b.

> 0:len=0x4 Change the length of extent 0 to 4.

> 1:off=0xa Change the logical block offset of extent 1 to 4.

> 0x43b:block/X Display the first four bytes of the contents of block 0x43b.

> 0x43b:block=0xdeadbeef
Set the contents of disk block number 0x43b to 0xdeadbeef. 0xdeadbeef may be truncated depending on the current *type*.

> 2050=0xffffffff Set the contents of address 2050 to 0xffffffff. 0xffffffff may be truncated depending on the current *type*.

> 1c92434="this is some text"
Place the ASCII for the string at 1c92434.

> 2:inode:0:db:block,*?d
Change the current inode to 2. Take the first block associated with this (root) inode and display its contents as directory entries. It stops prematurely if the EOF is reached.

SEE ALSO

fsck(1M), dir(4), efs(4), inode(4).

NAME

fsstat – report filesystem status

SYNOPSIS

fsstat special_file

DESCRIPTION

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

SEE ALSO

efs(4).

DIAGNOSTICS

The command has the following exit codes:

0 The filesystem is not mounted and appears O.K., except for root, where 0 means mounted and O.K.

1 The filesystem is not mounted and needs to be checked.

2 The filesystem is mounted.

3 The command failed.

NAME

ftp – Internet file transfer program

SYNOPSIS

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

DESCRIPTION

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

The client host with which *ftp* is to communicate can be specified on the command line. If this is done, *ftp* immediately attempts to establish a connection to an FTP server on that host; otherwise, *ftp* enters its command interpreter and awaits instructions from the user. When *ftp* is awaiting commands from the user, the prompt **ftp>** is provided to the user. The following commands are recognized by *ftp*:

! [*command* [*args*]]
Invoke an interactive shell on the local machine. If there are arguments, the first is taken to be a command to execute directly, with the rest of the arguments as its arguments.

$ *macro-name* [*args*]
Execute the macro *macro-name* that was defined with the **macdef** command. Arguments are passed to the macro unglobbed.

account [*passwd*]
Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is included, the user is prompted for an account password in a non-echoing input mode.

append *local-file* [*remote-file*]
Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local filename is used in naming the remote file after being altered by any *ntrans* or *nmap* setting. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII. This is the default type if *ftp* cannot determine the type of operating system running on the remote machine or the remote operating system is not UNIX.

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

binary Set the file transfer *type* to support binary image transfer. This is the default type if *ftp* can determine that the remote machine is running UNIX.

bye Terminate the FTP session with the remote server and exit *ftp*. An end of file also terminates the session and exits.

case Toggle remote computer filename case mapping during **mget** commands. When **case** is **on** (default is **off**), remote computer filenames with all letters in upper case are written in the local directory with the letters mapped to lower case.

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

cdup Change the remote machine working directory to the parent of the current remote machine working directory.

chmod *mode file-name*
Change the permission modes for the file *file-name* on the remote system to *mode*.

close Terminate the FTP session with the remote server, and return to the command interpreter. Any defined macros are erased.

cr Toggle carriage return stripping during ascii type file retrieval. Records are denoted by a carriage return/linefeed sequence during ascii type file transfer. When **cr** is **on** (the default), carriage returns are stripped from this sequence to conform with the UNIX single linefeed record delimiter. Records on non-UNIX remote systems can contain single linefeeds; when an ascii type transfer is made, these linefeeds can be distinguished from a record delimiter only when **cr** is **off**.

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

debug [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified, it is used to set the debugging level. When debugging is **on**, *ftp* prints each command sent to the remote machine, preceded by the string **-->**.

dir [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory*, and, optionally, placing the output in *local-file*. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving **dir** output. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, or *local-file* is –, output comes to the terminal.

disconnect
A synonym for **close**.

form *format*
Set the file transfer *form* to *format*. The default format is **file**.

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

glob Toggle filename expansion for **mdelete**, **mget** and **mput**. If globbing is turned **off** with **glob**, the filename arguments are taken literally and not expanded. Globbing for **mput** is done as in **csh**(1). For **mdelete** and **mget**, each remote filename is expanded separately on the remote machine and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file: the exact result depends on the foreign operating system and FTP server, and can be previewed by doing:

mls *remote-files* **-**

Note: **mget** and **mput** are not meant to transfer entire directory subtrees of files. That can be done by transferring a **tar**(1) archive of the subtree (in binary mode).

hash Toggle hash-sign (**#**) printing for each data block transferred. The size of a data block is 1024 bytes.

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

idle [*seconds*]
Set the inactivity timer on the remote server to *seconds* seconds. If *seconds* is omitted, the current inactivity timer is printed.

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

ls [*remote-directory*] [*local-file*]
Print a listing of the contents of a directory on the remote machine. The listing includes any system-dependent information that the server chooses to include; for example, most UNIX systems produce output from the command **ls –lA**. (See also **nlist**.) If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving **ls** output. If no local file is specified, or if *local-file* is –, the output is sent to the terminal.

macdef *macro-name*
Define a macro. Subsequent lines are stored as the macro *macro-name*; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros. Macros remain

defined until a **close** command is executed. The macro processor interprets **$** and \ as special characters. A **$** followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A **$** followed by an **i** signals that macro processor that the executing macro is to be looped. On the first pass **$i** is replaced by the first argument on the macro invocation command line, on the second pass it is replaced by the second argument, and so on. A \ followed by any character is replaced by that character. Use the \ to prevent special treatment of the **$**.

mdelete [*remote-files*]
Delete the *remote-files* on the remote machine.

mdir *remote-files local-file*
Like **dir**, except multiple remote files can be specified. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving **mdir** output.

mget *remote-files*
Expand the *remote-files* on the remote machine and do a **get** for each filename thus produced. See **glob** for details on the filename expansion. Resulting filenames are then processed according to *case*, *ntrans*, and *nmap* settings. Files are transferred into the local working directory, which can be changed with **lcd** *directory*; new local directories can be created with **! mkdir** *directory*.

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

mls *remote-files local-file*
Like **nlist**, except multiple remote files can be specified, and the *local-file* must be specified. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving **mls** output.

mode [*mode-name*]
Set the file transfer *mode* to *mode-name*. The default mode is **stream** mode.

modtime *file-name*
Show the last modification time of the file on the remote machine.

mput *local-files*
Expand wild cards in the list of local files given as arguments and do a **put** for each file in the resulting list. See **glob** for details of filename expansion. Resulting filenames are then be processed according to *ntrans* and *nmap* settings.

newer *file-name*
Get the file only if the modification time of the remote file is more recent that the file on the current system. If the file does not exist on the current system, the remote file is considered *newer*. Otherwise, this command is identical to **get**.

nlist [*remote-directory*] [*local-file*]
Print a list of the files of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* prompts the user to verify that the last argument is indeed the target local file for receiving **nlist** output. If no local file is specified, or if *local-file* is –, the output is sent to the terminal.

nmap [*inpattern outpattern*]
Set or unset the filename mapping mechanism. If no arguments are specified, the filename mapping mechanism is unset. If arguments are specified, remote filenames are mapped during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, local filenames are mapped during **mget** commands and **get** commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. The mapping follows the pattern set by *inpattern* and *outpattern*. *inpattern* is a template for incoming filenames (which may have already been processed according to the **ntrans** and **case** settings). Variable templating is accomplished by including the sequences **$1**, **$2**, ..., **$9** in *inpattern*. Use \ to prevent this special treatment of the **$** character. All other characters are treated literally, and are used to determine the **nmap** *inpattern* variable values. For example, given *inpattern* $1.$2 and the remote filename **mydata.data**, $1 would have the value **mydata**, and $2 would have the value **data**. The *outpattern* determines the resulting mapped filename. The sequences **$1**, **$2**, ..., **$9** are replaced by any value resulting from the *inpattern* template. The sequence **$0** is replace by the original filename. Additionally, the sequence **[***seq1***,***seq2***]** is replaced by *seq1* if *seq1* is not a null string; otherwise it is replaced by *seq2*. For example, the command **nmap $1.$2.$3 [$1,$2].[$2,file]** would yield the output filename **myfile.data** for input filenames **myfile.data** and **myfile.data.old**, **myfile.file** for the input filename **myfile**, and **myfile.myfile** for the input filename **.myfile**. Spaces can be included in *outpattern*, as in this example:

```
nmap $1 |sed "s/  *$//" > $1
```

Use the \ character to prevent special treatment of the **$**, **[**, **]**, and **,** characters.

ntrans [*inchars* [*outchars*]]
Set or unset the filename character translation mechanism. If no arguments are specified, the filename character translation mechanism is unset. If arguments are specified, characters in remote filenames are translated during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during **mget** commands and **get** commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. Characters in a filename matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the character is deleted from the filename.

open *host* [*port*]
Establish a connection to the specified *host* FTP server. An optional port number can be supplied, in which case, *ftp* attempts to contact an FTP server at that port. If the *auto-login* option is on (default), *ftp* also attempts to automatically log the user in to the FTP server (see below).

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

proxy *ftp-command*
Execute an *ftp* command on a secondary control connection. This command allows simultaneous connection to two remote FTP servers for transferring files between the two servers. The first **proxy** command should be an **open**, to establish the secondary control connection. Enter the command **proxy ?** to see other *ftp* commands executable on the secondary connection. The following commands behave differently when prefaced by **proxy**:

open Does not define new macros during the auto-login process.

close Does not erase existing macro definitions.

get and **mget** Transfer files from the host on the primary control connection to the host on the secondary control connection.

put, **mput**, and **append**
Transfer files from the host on the secondary control connection to the host on the primary control connection.

Third party file transfers depend upon support of the FTP protocol PASV command by the server on the secondary control connection.

put *local-file* [*remote-file*]
Store a local file on the remote machine. If *remote-file* is left unspecified, the local filename is used after processing according to any *ntrans* or *nmap* settings in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

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

quit A synonym for **bye**.

quote *arg1 arg2 ...*
The arguments specified are sent, verbatim, to the remote FTP server.

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

reget *remote-file* [*local-file*]
Reget acts like get, except that if *local-file* exists and is smaller than *remote-file*, *local-file* is presumed to be a partially transferred copy of *remote-file* and the transfer is continued from the apparent point of failure. This command is useful when transferring very large files over networks that are prone to dropping connections.

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

remotestatus [*file-name*]
With no arguments, show status of remote machine. If *file-name* is specified, show status of *file-name* on remote machine.

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

reset Clear reply queue. This command re-synchronizes command/reply sequencing with the remote FTP server. Resynchronization may be necessary following a violation of the FTP protocol by the remote server.

restart *marker*
Restart the immediately following **get** or **put** at the indicated *marker*. On UNIX systems, marker is usually a byte offset into the file.

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

runique Toggle storing of files on the local system with unique filenames. If a file already exists with a name equal to the target local filename for a **get** or **mget** command, a **.1** is appended to the name. If the resulting name matches another existing file, a **.2** is appended to the original name. If this process continues up to **.99**, an error message is printed, and the transfer does not take place. The generated unique filename is reported. Note that **runique** does not affect local files generated from a shell command (see below). The default value is **off**.

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

sendport
Toggle the use of PORT commands. By default, *ftp* attempts to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when performing multiple file transfers. If the PORT command fails, *ftp* uses the default data

port. When the use of PORT commands is disabled, no attempt is made to use PORT commands for each data transfer. This is useful for certain FTP implementations that do ignore PORT commands but, incorrectly, indicate they've been accepted.

site *arg1 arg2 ...*
The arguments specified are sent, verbatim, to the remote FTP server as a SITE command.

size *file-name*
Return size of *file-name* on remote machine.

status Show the current status of *ftp*.

struct [*struct-name*]
Set the file transfer *structure* to *struct-name*. By default **stream** structure is used.

sunique
Toggle storing of files on remote machine under unique filenames. Remote FTP server must support FTP protocol STOU command for successful completion. The remote server reports a unique name. Default value is **off**.

system Show the type of operating system running on the remote machine.

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

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer *type* to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

umask [*newmask*]
Set the default umask on the remote server to *newmask*. If *newmask* is omitted, the current umask is printed.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified and the server requires it, *ftp* prompts the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user is prompted for it. If an account field is specified, an account command is relayed to the remote server after the login sequence is completed if the remote server did not require it for logging in. Unless *ftp* is invoked with **auto-login** disabled, this process is done automatically on initial connection to the FTP server.

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

? [*command*]
A synonym for help.

Command arguments that have embedded spaces can be quoted with quote (") marks.

ABORTING A FILE TRANSFER

To abort a file transfer, use the terminal interrupt key (usually <Ctrl-c>). Sending transfers are immediately halted. Receiving transfers are halted by sending a FTP protocol ABOR command to the remote server and discarding any further data received. The speed at which this is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, an **ftp>** prompt does not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence is ignored when *ftp* has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode can result from the ABOR processing described above or from unexpected behavior by the remote server, including violations of the FTP protocol. If the delay results from unexpected remote server behavior, the local *ftp* program must be killed by hand.

FILE NAMING CONVENTIONS

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

1. If the filename – is specified, the **stdin** (for reading) or **stdout** (for writing) is used.

2. If the first character of the filename is **|**, the remainder of the argument is interpreted as a shell command. *ftp* then forks a shell, using *popen*(3S) with the argument supplied, and reads (writes) from the stdout (stdin). If the shell command includes spaces, the argument must be quoted; for example, **"| ls –lt"**. A particularly useful example of this mechanism is: **dir | more**.

3. Failing the above checks, if globbing is enabled, local filenames are expanded according to the rules used in the *csh*(1) **glob** command. If the *ftp* command expects a single local file (for example, **put**), only the first filename generated by the globbing operation is used.

4. For **mget** commands and **get** commands with unspecified local filenames, the local filename is the remote filename, which can be altered by a **case**, **ntrans**, or **nmap** setting. The resulting filename can then be altered if **runique** is on.

5. For **mput** commands and **put** commands with unspecified remote filenames, the remote filename is the local filename, which can be altered by a **ntrans** or **nmap** setting. The resulting filename can then be altered by the remote server if **sunique** is on.

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters that can affect a file transfer. The *type* can be one of **ascii**, **image** (binary), **ebcdic**, and **local byte size** (for PDP-10's and PDP-20's mostly). *ftp* supports the ascii and image types of file transfer, plus local byte size 8 for **tenex** mode transfers.

ftp supports only the default values for the remaining file transfer parameters: *mode*, *form*, and *struct*.

OPTIONS

Options can be specified at the shell command line. Several options can be enabled or disabled with *ftp* commands.

−v (**verbose on**) Forces *ftp* to show all responses from the remote server, as well as report on data transfer statistics.

−n Restrains *ftp* from attempting **auto-login** upon initial connection. If auto-login is enabled, *ftp* checks the *.netrc* file (see below) in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* prompts for the remote machine login name (default is the user identity on the local machine), and, if necessary, prompt for a password and an account with which to login.

−i Turns off interactive prompting during multiple file transfers.

−d Enables debugging.

−g Disables filename globbing.

THE .NETRC FILE

The *.netrc* file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they can be separated by spaces, tabs, or newlines:

machine *name* Identify a remote machine name. The auto-login process searches the *.netrc* file for a **machine** token that matches the remote machine specified on the *ftp* command line or as an **open** command argument. Once a match is made, the subsequent *.netrc* tokens are processed, stopping when the end of file is reached or another **machine** or a **default** token is encountered.

default This is the same as **machine** *name* except that **default** matches any name. There can be only one **default** token, and it must be after all **machine** tokens. This is normally used as:

```
default login anonymous password user@site
```

thereby giving the user *automatic* anonymous *ftp* login to machines not specified in *.netrc*. This can be overridden by using the **–n** flag to disable auto-login.

login *name* Identify a user on the remote machine. If this token is present, the auto-login process initiates a login using the specified name.

password *string* Supply a password. If this token is present, the auto-login process supplies the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the *.netrc* file for any user other than *anonymous*, *ftp* aborts the auto-login process if the *.netrc* is accessible by anyone besides the user (see below for the proper protection mode.)

account *string* Supply an additional account password. If this token is present, the auto-login process supplies the specified string if the remote server requires an additional account password, or the auto-login process initiates an ACCT command if it does not. Note that if this token is present in the *.netrc* file, *ftp* aborts the auto-login process if the *.netrc* is accessible by anyone besides the user (see below for the proper protection mode).

macdef *name* Define a macro. This token functions like the *ftp* **macdef** command functions. A macro is defined with the specified name; its contents begin with the next *.netrc* line and continue until a null line (consecutive newline characters) is encountered. If a macro named *init* is defined, it is automatically executed as the last step in the auto-login process.

The error message

```
Error: .netrc file is readable by others.
```

means the file is ignored by *ftp* because the file's *password* and/or *account* information is unprotected. Use

```
chmod go-rwx .netrc
```

to protect the file.

SEE ALSO

ftpd(1M).

BUGS

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of carriage returns in the 4.2BSD UNIX ascii-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2BSD servers using the ascii type. Avoid this problem by using the binary image type.

NAME

fx – disk utility

SYNOPSIS

fx [**–l** logfile] [**–r** maxretries] [**–x**] [drivetype[**(**ctrlr**,**unit**)**]]

fx **–c** **–x** [**–l** logfile] [**–r** maxretries] drivetype**(**ctrlr**,**unit**)**
[**VERIFY**] [**INITIALIZE**] [**FORMAT**]

DESCRIPTION

fx is an interactive, menu-driven disk utility. It detects and maps out bad blocks on a disk. It also displays information stored on the label of the disk, including partition sizes, disk drive parameters, and the volume directory.

An *expert* mode, available by invoking with the **–x** flag, provides additional functions normally used during factory set-up or servicing of disks, such as formatting the disk and creating or modifying the disk label or drive parameters.

Warning: Unless you are very familiar with the parameters and partitions of your disks, you are **strongly** advised not to invoke the expert mode of *fx*. A mistake in expert mode can destroy all the data on the disk. When this option is used, *fx* also warns of discrepancies between the disk label and the parameters that are normally used for the drive and asks if you want to fix them. You should usually NOT change these unless you have all the data on the drive backed up and are prepared to restore it, because the changes frequently result in a different partition layout.

The **–r** *retries* option allows you to specify how many retries *fx* attempts when exercising the disk. If you have persistent soft errors, **–r 0** usually allows *fx* to find the bad sectors and spare them. For SCSI disks, see also the discussion of parameters in the section **LABEL MENU**.

The **–l** *logfile* option (in the IRIX command version only) causes *fx* to log disk errors, blocks that are forwarded, and other severe errors in the given file.

The **–c** option (in the IRIX command version only) is designed for the use of programs and scripts; the **–x** option must also be given. When used, the **VERIFY**, **INITIALIZE**, or **FORMAT** (or any combination) options must be given at the end of the command line, and the full drive specification must be given on the command line. In this mode, no keyboard input (except keyboard interrupts) is accepted, and any error causes the program to exit with a non-zero value, following an error message. A warning message is printed at startup that destructive operations will follow, with no subsequent confirmation required. Additionally, it is considered a fatal error if the drive contains any mounted filesystems, or is part of a mounted logical volume filesystem. The **VERIFY** option is the equivalent of **/exercise/complete –a**, and overwrites any existing data on the drive. The **INITIALIZE** option creates only the volume header and partition table; this is the minimum that needs to be done for a disk drive to be usable. The **FORMAT** option is the equivalent of **format** with the current parameters (all data on the drive is destroyed). All the above options create a new partition table (suitable for an option disk) and volume header, if necessary.

USING FX

There are two versions of *fx*. One runs in the standalone environment and must be used when the system disk is modified; it can be used for most other purposes as well, but may be less convenient. The *jag* (VME SCSI), *rad* (SCSI-attached RAID), and *fd* (floppy) devices are not supported in the standalone version; the **–l** option is also not supported in standalone.

The other version runs as an IRIX command and is normally used by the superuser. While some features can be used by an ordinary user if the disk device permissions permit, other features (typically formatting and bad block management) have permission checks within the various drivers that can only be used by the super user. A notable exception is that as shipped, all floppy-related *fx* features can be used by any user. When used on a mounted disk, or a disk whose partitions are part of mounted logical volume, this version warns you not to do anything destructive, but does not otherwise prohibit it.

A copy of the standalone version is normally kept in */stand/fx* and can be invoked when the system is not running by giving the following command at the PROM Command Monitor:

```
boot stand/fx
```

A standalone *fx* is provided in the */stand* directory of CD-ROM discs containing software distributions with install tools, and can be invoked by the Command Monitor command:

`boot -f dksc(`*ctlr*,*unit*,8)sash*CPU* dksc(*ctlr*,*unit*,7)stand/fx.*CPU*

(Use *CPU*, only for IP17.) Other systems use the ARCS PROM (R4K Indigo, Indigo2, Indy, Onyx, Challenge):

`boot -f dksc(`*ctlr*,*unit*,8)sashARCS dksc(*ctlr*,*unit*,7)stand/fx.ARCS

For R8000 and other systems with 64-bit ARCS PROM (e.g., Power Challenge, Power Onyx, and Power Indigo^2):

`boot -f dksc(`*ctlr*,*unit*,8)sash64 dksc(*ctlr*,*unit*,7)stand/fx.64

where *ctlr* is the controller number (usually 0), *unit* is the SCSI id of the CD-ROM drive.

fx can also be booted from tape by giving this command at the PROM Command Monitor:

`boot -f tpsc(`*ctlr*,*unit*)fx.*CPU*

or

`boot -f tpsc()fx.`*ARCS*

When the standalone version is booted without the **–x** option, it prompts to see if you wish to use the expert mode, because it is not uncommon to forget to specify it.

The command version of *fx* is invoked by name like any IRIX command.

On invocation, *fx* prompts for a disk controller type, with a default of the root disk controller type. Recognized controller types are *dksc* for SCSI drives, *rad* for SCSI attached RAID drives, *jag* for SCSI drives connected to the Jaguar VME SCSI controller, and *fd* for floppy drives. Note that *jag*, *fd*, and *rad* are not available in the standalone version and that not all types of systems support all of the above drive types.

Controller number is normally 0 unless your system has more than one controller and you wish to work on disks attached to an additional controller (1, 2, and so on). Drive number depends on controller type. Non-SCSI drives are numbered from 0 to 1 (or 0 to 3, if the controller can handle 4 drives), with drive 0 on controller 0 normally used as the root disk. SCSI drives are numbered from 1 to 7 or from 1 to 15 depending on the type of system, with drive 1 on controller 0 normally used as the root disk. *fx* next prompts for the drive type, with a default of the drive type stored in the disk label.

The controller type, controller number, and drive number as well as drive type can be given as command line parameters, bypassing the interactive questions just described. The format is (drive_type is unused for SCSI drives):

```
fx "controllertype(controller_number, drive_number)"
```

For example:

```
fx "dksc(0,1)"
```

The quotes are necessary in the first argument in the command version, because parentheses are shell special characters, and in the second because the drive name contains a space. For floppy disk drives, you are also prompted for the density to use.

Once controller type, controller number, drive number and drive type are selected, *fx* issues a diagnostic command to the drive. For SCSI drives, the drive information from the inquiry command is displayed, including the firmware revision; for other drive types, the previously assigned type from the volume header is displayed. A controller or drive self test is performed, followed by sanity checks on the partition layout. If any 'major' differences are found, you are asked if you want to use the existing values. It is almost always correct to keep the existing values, unless you are going to initialize the disk anyway.

If it appears that no valid volume header is present, *fx* asks if you want to use the defaults; you can answer **no** if you plan to set up custom parameters or partitions.

fx then enters its main menu. Menu items can be selected by typing the least unambiguous prefix (the portion included between **[** and **]**) or the full name. A menu item can be an action (for example, **exit**) or the name of a submenu (for example, **badblock**). Submenus have a trailing **/** to indicated that they are submenus.

Selecting a submenu name causes that submenu to be displayed, and items from it can be selected. To return to a parent menu from a submenu, enter two dots (**..**). The menus are organized as a hierarchy, so you can go up two levels by typing **../..**, or use a command several levels down by separating each level by a **/**. By typing a command pathname, such as

```
/label/show/partition
```

a command can be executed from any point in the menu hierarchy. Similarly, typing the full pathname of any menu moves you to that menu (this includes typing **/** for the top level).

To obtain help for the items on the current menu, enter a question mark (**?**) at the prompt. Many of the functions listed below have options to modify their actions; to obtain more information about them than the summary, enter **?** *item* where *item* can be either the least unambiguous prefix, or the full name. Most of the (non-default) options are not listed in this document.

To exit from *fx*, select **exit** at the main menu; a shorthand for exiting from any level is **/exit**. Entering **/..** from any menu allows you to select a different disk without having to exit and restart; the normal prompts occur if modified parameters are not yet committed to disk.

Once the main menu is reached, *fx* catches interrupts: an interrupt stops any operation in progress but does not terminate *fx* itself. The current operation executing in the disk driver (if any) completes first; this is most notable when formatting a SCSI disk, because that is a single operation lasting many minutes.

FX PROMPTS

A general note about prompts: when a prompt with the word **no** or the word **yes** appears at the end, simply pressing <Enter> accepts that value. For other prompts that ask a question, you must answer either **yes** or **no**. For prompts requesting numeric values, you can usually reply with a decimal number, or a hex number (a leading 0x). If a number is displayed at the end of the prompt, pressing <Enter> accepts that value. It is usually the current value, although it is sometimes a reasonable default.

In many cases, if you are unsure of what your choices are, typing a **?** gives you a short description of your choices.

TOP LEVEL MENU

The top level *fx* menu contains the following choices:

exit Exits from *fx*. If changes have been made to the copy *fx* keeps of the disk label and this has not been written to the disk, a prompt gives the option to write it to disk.

badblock Selects the menu of operations dealing with bad block handling.

debug Selects the menu of debug functions.

exercise Selects the menu of functions for analyzing the disk surface to find bad blocks.

label Selects the menu of functions for reading (and, in expert mode, modifying) the disk label.

repartition
Allows simple repartitioning of disks. A disk can be easily partitioned into a root (system) or option (all of usable disk in one partition) disk. The size of a single partition can be easily modified, with the adjacent partitions (if any) resized to match.

The remaining options appear only in expert mode.

auto Initializes a new disk. The disk is formatted, a label is created and written to it, and it is exercised to detect and map out bad blocks.

format Formats the disk, erasing all information on the disk. With SCSI disks, the whole disk is formatted in a single un-interruptible operation, lasting 5-25 minutes, depending size and type. (It is very rare that a low level format like this is necessary on a SCSI disk.) The *rad* drives should not be formatted using this command, see *raid*(1M) for more information.

BADBLOCK MENU

The list of bad blocks is maintained by the SCSI controller/formatter hardware on the disk drive; it can be interrogated and altered but does not appear in the user-readable part of the disk. Note that *rad* drives do not support badblock forwarding.

The **badblock** menu contains the following choices:

addbb Allows new bad blocks to be added to the badblock list. Blocks can be identified either by a single blocknumber or as cylinder/head/sector. To terminate adding bad blocks, enter two dots (**..**); this returns to the **badblock** menu. In the SCSI case, an entered bad block is immediately inserted in the on-disk list maintained by the SCSI controller/formatter. There is no way to remove an added bad block from a SCSI disk without reformatting the whole disk.

You are asked if you want to try and preserve the data. If the disk contains valuable data, answer **yes**; if the disk is blank, answer **no**. There are two applications for this command: to allow any mistake made during initial bad block entry to be corrected, and to move data back off a replacement track that is also going bad.

In the second case, be certain to use *dd*(1M) to save the data onto another disk, delete the badblock forwarding (**deletebb** orig), add the replacement track to the badblock list (**addbb** replacement), update the forwarding (**forward**), mark the original block as bad (**addbb** orig), update the forwarding again (**forward**), and use *dd*(1M) to replace the saved data. This procedure is necessary to ensure data integrity, because data may be corrupted when moving it back onto the original defective track.

showbb Displays the current badblock list. It is obtained by interrogating the SCSI drive. In this case, usually the physical location of the bad sectors is displayed. Some SCSI drives can also display the logical block number (**showbb –l**) or the bytes from index format (**showbb –b**). If new bad blocks develop during the life of the system, it is necessary to add these new bad blocks to the badblock list.

Typically, the disk driver prints error messages on the console when it encounters a bad block. These messages are also normally logged to the system log file */var/adm/SYSLOG*. The error messages gives the location of the bad block, either as a single block number or as cylinder, head, and sector (in a form such as chs: 123/4/5), depending on the controller type. The disk is identified by its special filename; see *dks*(7M), *jag*(7M), or *rad*(7M)

The SCSI disk driver prints bad block numbers relative to the start of the partition it is accessing, as well as the absolute block number. It is the absolute block number that must be used when adding a bad block.

Note: *fx* attempts to save data when mapping out bad blocks by re-reading the old data a number of times. In all cases, it is strongly recommended to make a backup of the disk before proceeding with any bad block operations. Bad block mapping is NOT supported for floppy disk drives.

To manually map out a bad block, follow the procedures below. Unless you are completely sure that a particular block or track is bad, it is often a good idea to use the **exercise** function to locate and automatically map out the bad blocks. In some cases, a bad block may be reported that was the first block of a read or write request and not the block that is actually bad. For this reason, the exercising routines attempts to read each block in a failed I/O individually to find the bad blocks.

Persistent soft errors may not be found by the exerciser and may require using the manual procedure. For SCSI drives, you may wish to reduce the number of retries performed by the drive itself to 0, if the drive supports it, so that *fx* is more likely to find and forward the bad block. See the section on parameters. The default exerciser function is to do a read-only scan of the entire disk surface. The exercising method only adds blocks that are unrecoverable. Using the **badblock** menu, you can add any block, whether it is bad or not. For SCSI drives, there is no way to remove a block from the badblock list without re-formatting the drive. Thus, the data in any block that is mistakenly entered may be permanently lost, if the original data could not be read. A read-only exercise pass may not work if the block fails on writes only, and the disk contains important data, so that a write-read-compare pass isn't practical. In this case, you may need to manually map the bad blocks. However, if the disk is backed up and can be restored after the exercise is complete, a write-compare exercise pass finds and automatically maps bad blocks.

The procedure for forwarding bad blocks is divided into two parts: for SCSI disks (both *dksc* and *jag*, but not *rad*) and for other types. SCSI disks are much simpler, because the badblock map is maintained by the drive itself, rather than by the driver. *rad* devices do not support badblock forwarding.

To enter new bad blocks, select the **addbb** item. Then enter the location of the bad block (*fx* accepts either a single blocknumber or a cylinder/head/sector specification). More than one bad block can be entered. When you have finished entering, terminate the entries by entering two dots (**..**). The updated badblock list must be saved to disk and the new bad blocks mapped out. Select the **forward** option on the **badblock** menu to do this.

For SCSI disks, bad blocks are mapped out as soon as they are entered by the **addbb** function. Nothing more needs to be done. All SCSI bad blocks are entered by logical block number relative to the start of the disk, using the **addbb** function. (Driver messages about bad blocks typically give two numbers, where the smaller one is relative to the start of a partition, and the larger is relative to the start of the disk.) Enter as many bad blocks as you want, one per line, ending the list by typing **..** on a line by itself. The **showbb** function displays the complete list of bad blocks. The **–m** option can be used to show only the manufacturer's bad block list in one of several formats, which vary from drive to drive. The default and the most common format is to display by cyl/head/sec, even though blocks are entered by logical block number.

All disk error messages are logged to the system log */var/adm/SYSLOG* by default. You should examine the log periodically. If the same blocks show up repeatedly, you should add them to the badblock list with the **exercise** method. If necessary, use the **badblock** *menu. It is best to replace a block that is going bad before it becomes unreadable.*

FX LABEL FUNCTIONS

fx can display the information in the various parts of the disk label. To do this, select the **label** option at the main menu. Then select the **readin** function, and select the parts of the label you wish to display. This reads in the information from the disk. This choice is not present for SCSI disks, because all of the drive related label information is read from the embedded drive controller. Return to the **label** menu and select **show**. The various parts of the label can be selected for display.

When expert mode is used, the label values can be changed. Some of the values that can be changed are also sent directly to the drive or controller. Changing some parameters may require reformatting the drive before it can be used.

LABEL MENU

This menu gives access to functions for displaying and, in expert mode, modifying information contained in the disk label. It contains the following items:

readin Allows part or all of the label to be read in from the disk. Selecting this item brings up a menu of the accessible parts of the label. (These are described in detail below.) Selecting a part causes that part to be read in from disk; there is also an **all** option, to read in all parts at once. Note that this is normally done automatically before the first menu is displayed.

show Allows display of parts of the label. As with **readin**, it brings up a menu of the label parts, allowing selection of the part to be displayed.

The remaining items appear only in expert mode, because they offer the possibility of changing data on the disk.

sync Writes the in-core copy of the disk label back to disk, as well as changing the parameters in the disk driver.

set Allows parts of the label to be modified. As for **readin**, it brings up a menu of the label parts, allowing selection of the part to be modified. The current values are given as the default in the prompts, so simply pressing <Enter> for every prompt leaves the values unchanged. For SCSI drives, the drive parameters are divided into **geometry** and **parameters** menus. Changes to the **geometry** values require that the drive be reformatted, while other changes do not require reformatting of the drive.

create Discards existing label information, and creates new label information. For SCSI drives, the information used to create the label is obtained from the drive by modesense commands. For other drive types, the information comes from tables compiled into *fx*, unless the **other** choice was selected for the drive type, in which case the user-entered data is used. This is normally used only for attempting to repair a damaged disk label (or to recover from major errors during **set**). As with **readin**, it brings up a menu of the label parts, allowing selection of the part to be worked on.

PARTS OF THE DISK LABEL

A disk label contains the following parts:

parameters

This is information used by the disk controller, such as disk geometry (for example, number of cylinders), and format information (for example, interleave). The parameters actually used depend on the type of controller. For SCSI disks, there is an additional menu called **geometry**, and changes to values on the **parameters** menu do not require a reformat of the drive; changes to those on the **geometry** menu do.

These values do not need to be changed in normal use. A full discussion of the disk controller and disk drive is beyond the scope of this document. The reader should refer to the manufacturer's documentation. Some parameters affect only the label, others are passed on to the controller or drive. For SCSI drives, the parameters are sent to the disk with the save-parameters bit set, so that they remain in force even if the system is restarted.

When exercising SCSI drives, and attempting to find blocks with soft errors, it may be advisable to set the number of retries performed by the drive to 0, so that intermittent errors can be found. You may also want to disable ECC error correction on the drive. Not all drives allow you to change the number of retries. If you do change it during the exercise pass, you probably want

to restore the old value before exiting.

geometry This menu exists only for SCSI disks. A change to any of the parameters on this menu requires reformatting the drive before it can be used. Not all drives support changing all geometry items. Some changes also affect drive capacity. For some drives this capacity change is reflected immediately in values read from the drive, while for others the new values are not returned until after the drive is formatted.

partitions The disk surface is divided for convenience into a number of different sections called partitions, which are used for various purposes. (See *intro*(7) for more details). When the operating system is accessing the disk, its drivers make the connection between the special filename and the physical disk partition, using information from the partition table in the disk label.

Even if not started in expert mode, the drive partitions can be displayed and changed by using the **repartition** menu; see the section **CHANGING DISK PARTITIONS**.

There can be up to 16 partitions on a disk, numbered 0 to 15 (though not all need be present). Partitions of 0 length (0 or -1 for backwards compatibility) are not normally displayed, because they are logically not present. Each partition is described by its starting block on the disk, its size in blocks, and a type indicating its expected use (for example, filesystem, disk label, swap, and so forth). The *MAKEDEV*(1M) program creates only the entries in */dev* for the SGI standard partitions (0, 1, 6, 7, vh (8), and vol (10)). If you create and use other partitions, you must create entries for them in */dev* with the *mknod*(1M) command. Older versions of IRIX removed these non-standard entries in */dev* on each software installation; this is no longer done.

For some drives with variable geometry, a partition layout is created that may result in either fewer or more *logical* cylinders being used than the drive actually has. Whether the value is larger or smaller depends on how the drive reports the number of sectors per track. It is sometimes reported as an average (that may be rounded up or down) and sometimes as the smallest number of sectors on any track.

sgiinfo This contains information kept for administrative purposes: the type of disk drive and its serial number. For labels created under IRIX 4.0, it also includes the version of *fx* that was used to create the label (and presumably to do the drive setup).

bootinfo This contains information used by the system PROMs during a normal system boot. It specifies the root partition, the name of the file on the root partition to boot, and the swap partition. Normal defaults for these are: unix for the bootfile, 0 for the root partition, and 1 for the swap partition.

directory Some system files are normally kept in the label area (volume header) on the disk. These are files used in standalone operations such as the standalone shell *sash* and sometimes the diagnostic program *ide*, depending on system type. The directory is a table in the label that enables these files to be located. The **show** submenu of the **label** menu allows the directory of

these files to be displayed.

The files in the disk label are manipulated by the use of *dvhtool*(1M). *fx* does not provide facilities for adding or deleting files. It writes the *sgilabel* file when it has changed and the user requests it. Also note that using **create/directory** clears the directory.

CHANGING DISK PARTITIONS

The top level menu **repartition** is provided in both the modes. In the expert mode, one additional function is provided. The **expert** function is simply an alternate method of reaching the **/label/set/partition** function, provided for ease of use. You need to use this function if you want to create or modify other partitions that are not normally used.

When this menu is entered, the current partition layout is displayed, as well as the total drive capacity. For all of the non-expert choices, you are asked if you really want to change the partition layout after choosing the function. You are warned that any existing data on the drive could be lost if the partitions are changed. Remember that you must normally use the *mkfs*(1M) command to create filesystems on partitions before you can install software or restore files onto them.

The **rootdrive** function creates a drive with the standard partitioning for a system (or root) drive. This function should be used if you are setting up a new drive or changing an option drive into a root drive.

The **optiondrive** function creates a drive with all of the usable area in a single partition (partition 7). Some space is still allocated to the volume label.

For dksc only, after a partitioning scheme is selected, you are prompted for the filesystem type (*xfs* or *efs*) that you wish to assign to the data partitions. For the **rootdrive** and **optiondrive** options, if *xfs* is selected, you are asked if you wish to create an XFS log partition. If you answer **yes**, *fx* makes partition 15 into a 4 megabyte *xfslog* partition. This is used by *xlv*(7M) for the log subvolume.

The **resize** function allows you to resize any of the standard partitions (root, swap, usr, xfslog, and entire). After you select this function, a message is shown, reminding you that after you finish resizing a partition, the other partitions are resized to match (if necessary). You are shown the changes and given a chance to reject them, before they are committed to the disk, unless no changes were made.

The default partition presented depends on whether the drive appears to be a system (root) drive, or an option drive. For option drives, the default is entire. For system drives, the default is the swap partition.

After choosing the partition, you are shown the current values for the partition and asked to choose the method of partitioning the drive. The choices are to resize by megabytes, blocks, cylinders, or as a percentage of the entire disk. The default is megabytes. Next you are shown the maximum allowable size and asked to enter the new size.

If you made a change, the new partition layout of the drive is shown. You are asked to confirm that you want to use it (with a default of **no**). If you accept it, the new partition layout is immediately written to the drive and driver.

EXERCISE MENU

This gives access to functions intended for surface analysis of the disk to find bad blocks. Only read-only tests are possible in normal (non-expert) mode. Destructive read-write tests are allowed in expert mode. For all choices except **random**, I/O is done one cylinder at a time, unless an error is found. If an error is found, the I/O is repeated one sector at a time to find the actual block that is bad.

For each unrecoverable error that is found, the failing block is added to the badblock list. The number of retries performed by *fx* itself defaults to 3. It may be set to any number, including 0, using the **–r** option. Most drivers, and some drives, do retries before reporting an error. For most SCSI drives, the number of retries performed can be set by using the **/label/set/parameters** menu. By using the **stoponerror** menu selection, you can have *fx* stop and ask you if you want to map the bad block. Whether you answer **yes** or **no**, you are asked if you want to continue exercising. This can be useful when trying to determine how many errors a disk has before you commit yourself to mapping the bad blocks.

butterfly Invokes a test pattern in which successive transfers cause seeks to widely separated areas of the disk. This stresses the head positioning system of the drive and sometimes finds errors that do not show up in a sequential test. It prompts for the range of disk blocks to exercise, number of scans to do, and a test modifier. Each of the available test patterns can be executed in a number of different modes (read-only, read-write, and so on) that are described below.

errlog Prints the total number of read and write errors that have been detected during a preceding exercise, showing both soft and hard errors. If the **–l** option is used, the blocks on which errors occurred are also reported. Soft errors are those errors for which a driver reported an error, but *fx* was able to successfully complete the I/O on a retry. Blocks with soft errors are not forwarded.

random Invokes a test pattern in which the disk location of successive transfers is selected randomly. It is intended to simulate a multiuser load. Like the **butterfly** test, it prompts for range of blocks to exercise, number of scans, and modifier. This does random sized I/Os (from one block to the cylinder size) as well as seeking to random locations on the disk. It is useful for finding problems on drives with seek problems and with errors in the caching logic or hardware.

sequential
Invokes a test pattern in which the disk surface is scanned sequentially. As with the **butterfly** test, it prompts for: range of blocks to exercise, number of scans, and modifier.

stop_on_error
Toggles whether *fx* proceeds automatically when errors are detected. The default is automatic. If stop is set, you are asked on each error whether you want to continue or not. If you continue, you are asked if you want to add the failing block to the badblock list. This can be useful if you

want to find all the failing bad blocks but not actually add them to the badblock list.

The following items appear only in expert mode, because they are concerned with destructive (write) tests.

settestpat Allows you to specify the pattern of data that to be used in tests that write to the disk to be created. Up to 4K bytes of pattern can be set, byte by byte. Each byte can be entered as a decimal or hex value (with a leading 0x). Enter **..** when you are done entering the pattern. The pattern is repeated as many times as necessary to fill the buffer. The default is a random pattern 1023 bytes long ensuring that few, if any sectors have the same data. When used with the write-compare test, this helps find drives that have hardware or firmware problems causing them to write data to the wrong location on the drive.

showtestpat
Displays the pattern of data that is used in tests that write to the disk. This can be changed with **settestpat**.

complete Causes a write-and-compare sequential test to be run on the entire disk area; all data on the drive is lost.

The **butterfly**, **random**, and **sequential** tests prompt for a modifier that determines the type of transfer that occurs during the test patterns. Possible modifiers are:

rd-only Performs reads only. The value of read data is ignored. The test detects only the success or failure of the read operation.

rd-cmp Causes two reads at each location in the test pattern. The data obtained in the two reads is compared. If there is a difference, the blocks that differ are considered bad.

seek Causes each block in the test pattern to be read (no writes) separately. It is used to verify individual sector addressability. (This is a rather time-consuming operation!)

The following modifiers are presented and legal only in expert mode, because they cause writing to the disk, thereby destroying existing data. Be absolutely sure you have backed up any data you care about before using them. You are given one last chance to abort after you have specified all the parameters to use.

wr-only Performs writes only. Written data is not re-examined. The test detects only the success or failure of the write operation. Certain kinds of media errors cause write errors, but not read errors.

wr-cmp Performs a write, read, compare operation. If any of the three operations fail, the block is considered to be bad. Data miscompares are reported differently than I/O errors, but a data miscompare still causes the block with the miscompare to be added to the badblock list. This is the most thorough test and highly recommended after formatting a drive.

DEBUG FUNCTIONS

fx has a menu of disk debug functions. For safety reasons, most are not present in the normal (non-expert) mode, where only nondestructive functions are available. In the expert mode, disk blocks can be written as well as read. For SCSI disks, the drive parameters (modesense pages) can be displayed and individual bytes altered and sent to the disk via modeselect commands.

A function that can be useful is the ability to directly read and display the contents of any block on the disk. An internal memory buffer is provided as a source or destination for data; the contents of this buffer can be displayed and edited.

For SCSI drives, there are also functions to display the drive capacity, to display the modesense page values, and to allow setting of modeselect page values (as decimal, octal, or hex values, rather than symbolicly, as is normally done with the **label** functions).

cmpbuf Allows blocks of data in different areas of the buffer to be compared; written and read-back data, for example. It prompts for the starts of the two areas to be compared (relative to the beginning of the internal buffer) and for the length of comparison.

dumpbuf Allows display of the contents of the buffer. It prompts for start address (relative to beginning of buffer), length to display and display format: bytes, (2-byte) words, or (4-byte) longwords. Data is displayed in the hex format selected and also in character format with non-printable characters represented by dots.

editbuf Allows individual buffer locations to be modified in byte, 2-byte or 4-byte units.

fillbuf Allows sections of the buffer to be filled with a repeating pattern. It prompts for start location and length to fill and for a string of data to use as the fill pattern. (Unfortunately, only a string is accepted. It is not possible to enter hex data. The buffer can be cleared by entering a null string.)

number Accepts a decimal number, and prints it in octal and hex.

readbuf Allows disk blocks to be read into the internal buffer. It prompts for buffer address (relative to start of buffer), and number of blocks to read. Up to 100 blocks can be read in one operation. The disk block address from which the read occurs is maintained as an internal variable by *fx*. It can be set with the **seek** function.

seek Sets the internal *fx* variable that holds the source or destination blocknumber on disk for transfers between disk and the internal buffer. A prompt of the current value is given. It does not cause any I/O, just sets the block number for the next I/O.

The remaining functions appear only in expert mode, because they are either potentially destructive (for example, **writebuf**) or of little interest to the normal user.

writebuf Writes blocks from the internal buffer to the disk. It prompts for source buffer address and number of blocks to write. The disk address block for the write is taken from the internal *fx* variable set by **seek**, as for **readbuf**.

showcapacity
Appears only for SCSI drives. It shows the output of the SCSI readcapacity command. This can be used to verify that the partition layout chosen is valid (*fx* verifies this automatically, but it can still be useful to see this). Drives with variable geometry can have a partition layout that does not use all of the drive. The partitions should never extend past the value displayed by **showcapacity**. Note that after geometry on SCSI drives is changed, the drive may not report any capacity changes until after a format is done.

showpages
Appears only for SCSI drives. It shows which modesense pages (drive parameters) the drive supports, their length, and, with the **–l** option, their current values. The **–c** and **–d** options display the changeable and default values, respectively. This is sometimes useful when attempting to connect a drive that has features not already supported by *fx*.

setpage Appears only for SCSI drives. It allows you to set the values of a modeselect page (and optionally the block descriptor) on a byte by byte basis. As with other *fx* input, numbers are decimal by default, octal with a leading 0, or hex with a leading 0x. Trailing bytes not entered are treated as 0. The values are masked with the changeable values; the masked values are displayed before they are set. There are **no** sanity checks on the values entered (other than that they must fit in a byte). Therefore it is possible to render a drive unusable by changing values this way. This function is intended for those who understand the meanings of the values in the modeselect pages, primarily when dealing with new types of drives. It is sometimes possible to recover from mistakes by doing **/label/creat/all**, following by **/format**.

INITIALIZING NEW DISKS

fx can be used to initialize disk drives that have not been previously formatted. The new drive to be initialized MUST be physically connected to the system.

Warning: Do not connect or disconnect non-RAID drives while the system is powered up, because this could damage the drive or controller. On SCSI drives, it could also cause the termination power fuse to fail (on having them; some have solid state replacements), resulting in apparently random SCSI errors.

The disk drives in a RAID brick can be removed and added while the system is up and accessing the RAID. Initialization of a RAID should be done using the RAID administrative utility *raid*(1M).

Take care that termination of the new drive is correct. This varies with the drive type and system type. On systems with SCSI drives and an external terminator pack, none of the drives should be terminated unless they are external to the system; in that case, only the device at the end of the SCSI bus should have terminators. Be sure that the drive ID does not conflict with that of any other drive connected to the same controller. For all systems shipped by SGI, the controller (host adapter) SCSI ID is 0. Many other

manufacturers' systems are shipped with the controller as ID 7, so be sure to check the ID when moving drives from one type of system to another.

With the new drive connected, bring the system back up to normal multiuser mode, and invoke *fx* in expert mode (the **–x** option). Enter the controller type and number, and the drive number for the new drive. For SCSI drives, the drive type is determined automatically by an inquiry operation on the drive.

SCSI drives determine all of the information about the drive by using the modesense command, after determining which modesense pages the drive supports. If the drive supports the SCSI 2 pages, they are used. Otherwise, the CCS extensions to SCSI 1 are assumed (as well as some defacto standard vendor-specific pages). If none of the geometry pages are supported, *fx* chooses some reasonable set of defaults, such that most disks should be able to be used to their full capacity. Use of drives not qualified by Silicon Graphics Inc., is not recommended.

Once drive type is identified, select the **auto** item on the main menu. This formats the drive, scans it for bad blocks, and places a label on it. On completion, exit from *fx*. The drive is now ready for use.

It is usually necessary to make filesystems on the drive and to mount these filesystems before the drive can be used. See *mkfs*(1M), *Add_disk*(1) (for SCSI dksc drives only), and *mount*(1M).

Note: Use of **auto** on SCSI drives formats the drive with the current drive parameters. Older versions used the manufacturer's default parameters, which did not always match the parameters as shipped by Silicon Graphics. The default parameters work, but may not give you the same drive capacity or performance as those shipped by Silicon Graphics.

FILES

/dev/rdsk/jag*, /dev/rdsk/dks*, /dev/rdsk/fds*, /dev/rdsk/rad*

SEE ALSO

Add_disk(1), MAKEDEV(1M), dvhtool(1M), mknod(1M), mount(1M), dks(7M), jag(7M), rad(7M), smfd(7M), vh(7M).

NAME

growfs – expand a filesystem

SYNOPSIS

growfs [**–s** size] special

DESCRIPTION

growfs expands an existing Extent Filesystem, see *efs*(4). The *special* argument is the pathname of the device special file where the filesystem resides. The filesystem must be unmounted to be grown, see *umount*(1M). The existing contents of the filesystem are undisturbed, and the added space becomes available for additional file storage.

If a **–s** *size* argument is given, the filesystem is grown to occupy *size* basic blocks of storage (if available).

If no *size* argument is given, the filesystem is grown to occupy all the space available on the device.

growfs is most often used in conjunction with logical volumes; see *lv*(7M) and *xlv*(7M). However, it can also be used on a regular disk partition, for example, when a partition has been enlarged while retaining the same starting block.

To grow XFS filesystems, use the *xfs_growfs*(1M) command.

PRACTICAL USE

Filesystems normally occupy all of the space on the device where they reside. In order to grow a filesystem, it is necessary to provide added space for it to occupy. Therefore there must be at least one spare new disk partition available.

Adding the space is done through the mechanism of logical volumes.

If the filesystem already resides on a logical volume, the volume is simply extended using *mklv*(1M).

If the filesystem is currently on a regular partition, it is necessary to create a new logical volume whose first member is the existing partition, with subsequent members being the new partitions to be added. Again, *mklv* is used for this.

In either case *growfs* is run on the logical volume device, and the expanded filesystem is available for use on the logical volume device.

DIAGNOSTICS

growfs expands only clean filesystems. If any problem is detected with the existing filesystem, the following error message is printed:

```
growfs: filesystem on <special> needs cleaning.
```

If a *size* argument is given, *growfs* checks that the specified amount of space is available on the device. If not, it prints the error message:

```
growfs: cannot access <size> blocks on <special>.
```

growfs works in units of the cylinder group size in the existing filesystem. To usefully expand the filesystem there must be space for at least one new cylinder group. Failing this, it prints the error message:

```
growfs: not enough space to expand filesystem.
```

COMPATIBILITY NOTE

growfs can expand a filesystem from any IRIX release, and filesystems can be expanded repeatedly. However, once a filesystem has been grown, it is NOT possible to mount it on an IRIX system earlier than release 3.3, and a pre-3.3 *fsck* does not recognize it.

SEE ALSO

mkfs(1M), mklv(1M), xfs_growfs(1M), lv(7M), xlv(7M).

NAME

hinv – hardware inventory command

SYNOPSIS

hinv [**–v**] [**–s**] [**–c** class] [**–t** type] [**–d** dev] [**–u** unit]

DESCRIPTION

hinv displays the contents of the system hardware inventory table. This table is created each time the system is booted and contains entries describing various pieces of hardware in the system. The items in the table include main memory size, cache sizes, floating point unit, and disk drives. Without arguments, the *hinv* command displays a one line description of each entry in the table.

The *hinv* options are:

–v Gives a more verbose description of some items in the table.

–c *class* Displays items from *class*. *classes* are **processor**, **disk**, **memory**, **serial**, **parallel**, **tape**, **graphics**, **network**, **scsi**, **audio**, **iobd**, **video**, **bus**, **misc**, **compression**, **vscsi**, and **display**.

–t *type* Displays items from *type*. *types* are **cpu**, **fpu**, **dcache**, **icache**, **memory**, **qic**, **a2**, and **dsp**.

–s When used with either the **–c** or **–t** options, **–s** suppresses output.

–d *dev* Restricts the output to a kind of device among those currently present in the system. The devices that currently support this option are **cdsio**, **aso**, **ec**, **et**, **ee**, **enp**, **fxp**, **ep**, **hy**, **ipg**, **xpi**, **fv**, **gtr**, **mtr**, and **atm**.

–u *unit* Requests information for a single device unit number for a kind of device specified with **–d**.

The *hinv* command, when used with the **–c** or **–t** options, exits with a value of 1 if no item of the specified class or type is present in the hardware inventory table. Otherwise, *hinv* exits with a value of 0.

NOTE

For many devices, the device is displayed in the inventory if the corresponding driver is not configured into IRIX.

SEE ALSO

lboot(1M), getinvent(3).

NAME

icrash – IRIX system crash analysis utility

SYNOPSIS

icrash [**–f** cmdfile] [**–r**] [**–v**] [**–w** outfile] [**–F**] namelist corefile

DESCRIPTION

icrash is a hands-on utility that generates detailed kernel information in an easy-to-read format. *icrash* also provides the ability to generate reports about system crash dumps created by *savecore*(1M). Depending on the type of system crash dump, *icrash* can create a unique report that contains information about what happened when the system crashed. *icrash* can be run on both live systems or with any *namelist* and *corefile* specified on the command line.

namelist contains symbol table information needed for symbolic access to the system memory image being examined. The default *namelist* is **/unix**, which is used when analyzing a live system. If the memory image being analyzed is from a system core dump (*vmcore.N.comp*), then *namelist* must be a copy of the unix file that was executing at the time (*unix.N*).

corefile is a file containing the system memory image. The default *corefile* is **/dev/mem**, which provides access to system memory when analyzing a live system. *corefile* can also be a pathname to a file (*vmcore.N.comp*) produced by the *savecore*(1M) utility.

The *icrash* command has the options listed below. By default, all information is sent to the standard output, unless the **-w** option is used:

–f *cmdfile* Specifies a *cmdfile* that contains a set of commands that *icrash* runs automatically.

–r Generates a standard report for the *namelist* and *corefile* specified. The reporting style differs depending on the type of system dump specified on the command line.

–v Prints the current version number.

–w *outfile* Writes any generated output to *outfile*.

–F Prints out field replacement unit information for any hardware problem that *icrash* can detect in the *namelist* and *corefile* specified. This option applies to CHALLENGE and Onyx systems only.

REPORT USAGE

In order to generate a crash report with *icrash* you must use the **–r** flag. For example:

```
icrash -r namelist corefile
```

This creates an **ICRASH CORE FILE REPORT** that prints to standard output. This report can be analyzed to determine what type of system failure has occurred and what exactly occurred for this crash dump. Note that you cannot use the **–r** flag against a live system.

INTERACTIVE USAGE

Input during an *icrash* session is of the form:

function [*argument* ...]

where *function* is one of the *icrash* functions described in the **FUNCTIONS** section of this reference page and *argument* is qualifying data that indicates which item or items of a particular kernel structure to print.

The following options are available to all *icrash* functions wherever they are semantically valid (see the **FUNCTIONS** section below).

−a Display all entries -- even those that are currently unallocated, on a free list, or have a reference count of zero.

−f Display additional (full) information for a structure.

−n Follow the links of related kernel structures to their logical conclusion. For example, when used with the **stream** command, information for the stdata structure for the stream is displayed, followed by information on each of the queue pairs on the stream.

−w *outfile* Redirect the output of a function to the named *outfile*.

All *icrash* functions can be piped with the command:

function [*argument* ...] **|** *shell_command*

Depending on the context of the function, numeric arguments are assumed to be in a specific radix. Counts are assumed to be decimal. Addresses are always hexadecimal (with or without a leading **0x**). Table address arguments larger than the size of the function table are interpreted as hexadecimal addresses; those smaller are assumed to be decimal slots in the table.

FUNCTIONS

Below is a list of functions that come with *icrash*. Please note that this list can change from one release to the next. To get a list of all the functions available within *icrash*, use the **help** command.

base *numeric_value* ...
Display a 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, hexadecimal; 0, octal; and 0b, binary.

curproc [**−f**] [**−w** *outfile*]
Display the proc table entry for the currently running process (or processes when there are multiple CPUs).

defproc [*slot_number*]
Set defproc (the default process) if process *slot_number* is indicated. Otherwise, display the current value of defproc. When *icrash* is run on a system core dump, defproc gets set automatically to the proc *slot_number* for dumpproc (the process running at the time of the PANIC). When *icrash* is run on a live systems, no default process is set. Note that it is possible, with a system core dump, for no default process to be set. If the PANIC occurs while a CPU is servicing an interrupt from an idle state, there is no current process.

Defproc is used by *icrash* in a number of ways. The **trace** command displays a trace for the default process if one is set. Also, the translation of certain kernel virtual addresses depend upon defproc being set. Currently these include the virtual address for the user block and kernel stack for a given process.

dis [**−w** *outfile*] *address* [*count*]
Display the disassembled code from address for *count* instructions. The default count is 1.

dump [**−d**] [**−o**] [**−x**] [**−B**] [**−D**] [**−H**] [**−W**] [**−w** *outfile*] *start_address* [*count*]
Display count values starting at kernel virtual address *start_address* in one of the following formats: decimal (**−d**), octal (**−o**), or hexadecimal (**−x**). The default format is hexadecimal, and the default count is 1.

eframe [**−w** *outfile*] *address*
Display the exception frame (containing a register dump, EPC, cause register, and status register) located at *address*.

etrace [**−f**] [**−w** *outfile*] *eframe* [*spage*]
Display a stack trace using the PC and SP found in the exception frame pointed to by eframe. If *spage* is provided, use that as the stack page address. Otherwise determine the address of the stack page using the stack pointer from the exception frame. Note that defproc must be set equal to the proc slot of the process being analyzed when spage is the address of the kernel stack or when spage is the CPU interrupt stack and there is a process running on that CPU.

eval [**−d**] [**−o**] [**−w** *outfile*] *equation*
Converts a simple equation to a single numeric value.

file [**−f**] [**−n**] [**−a**] [**−p** *proc_list*] [**−w** *outfile*] [*file_list*]
Display the file structure located at each virtual address included in *file_list*. If no addresses are specified, display the entire kernel file table. If the **−p** option is used, display all files opened by proc. Proc can be specific as a proc slot number, process PID (preceded by a **#**), or virtual address.

findsym [**−f**] [**−w** *outfile*] *address_list*
Locate the kernel symbol closest to each virtual address contained in *address_list*.

from2 *cmdfile*
Read in commands from *cmdfile* and run them as if typed from within *icrash*. Note that the **–w** option and **|** option are not valid commands on the command line, but can be placed into the cmdfile for execution.

fru [**–w** *outfile*]
Print out the field replacement unit analysis information from any core dump. Percentages are displayed based on the amount of confidence the fru module has in the particular board identified as being a problem. Multiple boards are identified where applicable.

fstype [**–w** *outfile*] [*vfssw_list*]
Display the vfssw structure for each virtual address included in *vfssw_list*. If no vfssw structures are specified, display the entire vfssw table.

help [**all** | *command_list*]
Display a description of the named functions, including syntax. The **all** option displays help information for every command.

history
Dump out the last 20 history commands. You can also use **!** to access the old commands (including **!!**, **!**–*N*, and so on)

inode [**–f**] [**–w** *outfile*] *inode_list*
Display the inode structure located at each virtual address included in *inode_list*.

inpcb [**–f**] [**–w** *outfile*] *inpcb_list*
Display the inpcb structure located located at each virtual address included in *inpcb_list*.

lastproc [**–f**] [**–w** *outfile*]
Display the proc table entry for the last running process (or processes when there are multiple CPUs).

mbstat
Dump out the mbuf statistics in the corefile.

mbuf [**–f**] [**–n**] [**–w** *outfile*] *mbuf_list*
Display the mbuf structure located at each virtual address included in *mbuf_list*.

outfile [*outfile*]
Set *outfile* (the file where all command output is sent) if *outfile* is indicated. Otherwise, display the current value of *outfile*.

pde [**−w** *outfile*] *pde_list*
Display the pde (page descriptor entry) structure located at each virtual address included in *pde_list*.

pfdat [**−f**] [**−a**] [**−w** *outfile*] *pfdat_list*
Display the pfdat structure located at each virtual address included in *pfdat_list*.

pfdathash [**−f**] [**−a**] [**−w** *outfile*] [*bucket_list*]
Display all pfdat structures in each of the page hash table buckets included in *bucket_list*. If no page hash table buckets are specified, display the pfdat structures in all pfdat hash table buckets. Items on *bucket_list* can consist of phash indexes (in decimal form) and/or hexadecimal virtual addresses (in any order).

pfind [**−f**] [**−w** *outfile*] *tag* [*pgno*]
Display all pfdat structures currently in the page hash table with tag. If *pgno* is specified, display only the pfdat structure that exactly matches *tag* and *pgno*.

pregion [**−f**] [**−n**] [**−a**] [**−p** *proc_list*] [**−w** *outfile*] *pregion_list*
Display the pregion structure located at each virtual address included in *pregion_list*. If the **−p** option is used, display all pregions allocated to proc. Proc can be specific as a proc slot number, process PID (following a **#**), or virtual address.

proc [**−f**] [**−a**] [**−n**] [**−w** *outfile*] [*proc_list*]
Display the proc structure for each entry in *proc_list*. If no entries are specified, display the entire proc table. Entries in *proc_list* can take the form of a proc table entry (slot number), process PID (following a **#**), or virtual address.

ptov [**−w** *outfile*] *address_list*
Display all possible virtual address mappings (K0, K1, and K2) for each entry in *address_list*. Entries in *address_list* can be a hexadecimal physical address or a PFN (following a **#**).

queue [**−f**] [**−n**] [**−w** *outfile*] *queue_list*
Display the queue structure located at each virtual address included in *queue_list*. If the next option (**−n**) is specified, a linked list of queues, starting with each specified queue then following the q_next field, is displayed.

quit Exit *icrash*. Note that **q** prompts for confirmation unless a **!** is appended to the command line.

region [**−f**] [**−n**] [**−p** *proc_list*] [**−w** *outfile*] *region_list*
Display the region structure located at each virtual address included in *region_list*. If the **−p** option is used, display all regions allocated to proc. Proc can be specific as a proc slot number, process PID (following a **#**), or virtual address.

rnode [**−f**] [**−a**] [**−w** *outfile*] *rnode_list*
Display the rnode structure for each virtual address included in *rnode_list*.

runq [**−f**] [**−n**] [**−w** *outfile*]
Display a list of processes currently on the run queue.

search [**−B**] [**−D**] [**−H**] [**−W**] [**−w** *outfile*] [**−m** *mask*] *pattern* [*address*] [*length*]
Locate contiguous bytes of memory that match the values contained in *pattern*, beginning at *address* for *length* bytes. Pattern consists of a string of from one to 256 hexadecimal digits (with no embedded spaces). For full word searches (the default), the first word of mask is anded (&) with each word of memory and the result compared against the first word in *pattern* (which is also anded with the first word of *mask*). If there is a match, subsequent words in memory are compared with their respective words in *pattern* (if there are any) to see if a match exists for the entire pattern. If the **−h** option is issued, the search is conducted on halfword boundaries. If the **−b** option is issued, the search is performed without regard to word or halfword boundaries. If a *mask* is not specified, *mask* defaults to all ones for the size of pattern.

The *address* can be specified as either a virtual address (K0, K1, K2, or KPTESEG), a physical address, or as a PFN (directly following a pound **#** sign). If no *address* is specified (or if the one specified does not map to a valid physical memory address), address defaults to the K0 address mapped to the start of physical memory (0x80000000 on most systems, 0x88000000 for IP17, IP22, and IP23 systems). An optional *length* parameter specifies the number of bytes to search. If *length* is not specified, it is set equal to the size of physical memory minus the starting physical address. Note that *length* can be specified ONLY when a *address* has been specified.

sema [**−f**] [**−n**] [**−w** *outfile*] [*sema_list*]
Display the sema_s structure located at each virtual address included in *sema_list*.

sh [*cmd*]
Escape to the shell with no arguments or execute the command entered. Note that it uses your SHELL environment variable to determine which shell to use.

sizeof *structure_names*
Dump out the size of a structure entered on the command line. The value returned is in bytes.

snode [**−f**] [**−a**] [**−w** *outfile*] *snode_list*
Display the snode structure located at each virtual address included in *snode_list*.

socket [**−f**] [**−n**] [**−w** *outfile*] [*socket_list*]
Display the socket structure for each virtual address included in *socket_list*. If no entries are specified, display all sockets that are currently allocated. If the next option (**−n**) is specified, a linked list of protocol control block structures associated with each socket is also displayed.

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

stream [**–f**] [**–a**] [**–n**] [**–w** *outfile*] [*stream_list*]
Display the stdata structure for each virtual address included in *stream_list*. If no entries are specified, display all streams that are currently allocated. If the next option (**–n**) is specified, a linked list of queues that are associated with the stream is also displayed.

string [**–w** *outfile*] *start_address* | *symbol* [*count*]
Display *count* strings of ASCII characters starting at *start_address* (or address for symbol).

strstat [**–f**] [**–w** *outfile*]
Display information about streams related resources (streams, queues, message blocks, data blocks, and zones).

strace [**–a**] [**–l**] [**–f**] [**–w** *outfile*] [*pc*] [*sp*] *spage* [*level*]
Display all complete and unique (and potentially valid) stack traces found in spage having level or more frames (the default value of level is 5). Alternately, attempts to display a stack trace starting at a particular PC and SP within spage. Or, when the **–l** option is specified, displays a list of all valid kernel code addresses contained in spage along with their location in the stack, source file and line number. If the **–a** option is specified, all traces of level or more frames are displayed. Note that defproc must be set equal to the proc slot of the process being analyzed when spage is the CPU interrupt stack and there is a process running on that CPU.

struct [**–f**] [**–n**] [**–w** *outfile*] *structure*
Dump out the size of a structure, or if the **–f** option is used, dump out the sizes and offsets of items in the structure.

swap [**–f**] [**–w** *outfile*] *swap_list*
Dump out the list of swap devices, including the vnodes that are represented. The number of pages, number of free pages, number of max pages, priority, and device are listed. The **–f** flag dumps out the name of the swapinfo entry as well.

symbol [**–f**] [**–w** *outfile*] *symbol_list*
Displays information about each kernel symbol included in *symbol_list*.

tcp [**–f**] [**–w** *outfile*] *tcpcb_list*
Display the tcpcb structure for each virtual address included in *tcpcb_list*.

tlb [**–w** *outfile*] [*cpu_list*]
Display TLB information for each CPU indicated in *cpu_list*. If no CPUs are indicated, TLB information for all CPUs is displayed.

trace [**−f**] [**−a**] [**−w** *outfile*] [*proc_list*]
Displays the kernel stack trace for each process included in *proc_list*.

uipcvn [**−f**] [**−n**] [**−w** *outfile*] [*socket_list*]
Walk the uipc_vnlist and search for each socket on *socket_list*. When a match is found print the vnode and socket structures. If no entries are specified, display all vnode/socket structures on uipc_vnlist. If the next option (**−n**) is specified, a linked list of protocol control block structures associated with each socket is also displayed.

unpcb [**−f**] [**−w** *outfile*] [*unpcb_list*]
Display the unpcb structure for each virtual address included in *unpcb_list*.

user [**−f**] [**−w** *outfile*] [*proc_list*]
Display the user structure for each entry in *proc_list*. If no entries are specified, display the user area for all active processes. Entries in *proc_list* can take the form of a proc table entry (slot number), process PID (following a **#**), or virtual address.

vfs [**−f**] [**−w** *outfile*] *vfs_list*
Display the vfs structure for each virtual address included in *vfs_list*.

vnode [**−f**] [**−a**] [**−n**] [**−w** *outfile*] *vnode_list*
Display the vnode structure for each virtual address included in *vnode_list*.

vtop [**−w** *outfile*] *address_list*
Display all possible virtual address mappings (K0, K1, and K2) for each virtual address in *address_list*.

zone [**−f**] [**−w** *outfile*] [*zone_list*]
Display information about zone memory allocation resources.

? [**−w** *file*]
Displays a list of available functions.

NOTES

Each version of *icrash* is specific to the OS release that it came from and does not necessarily work on any other OS release. Do not copy *icrash* to any other IRIX system unless the OS versions are identical (including patch levels).

Running *icrash* on a live system can sometimes generate random results, as the information being viewed is volatile at the time it is displayed.

The GNU readline library is used by *icrash* in interactive mode. Please see the GNU General Public License available from the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

NAME

ifconfig – configure network interface parameters

SYNOPSIS

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

DESCRIPTION

ifconfig is used to assign an address to a network interface and/or configure network interface parameters. *ifconfig* is invoked at boot time from */etc/init.d/network* to define the network address of each interface present on a machine; you can also use it once the system is up to redefine an interface's address or other operating parameters. The *interface* parameter is a string of the form ''name unit'', for example, **enp0**. (The **–i** option to *netstat*(1) displays the interfaces on the machine.)

Since an interface can receive transmissions in differing protocols, each of which may require separate naming schemes, it is necessary to specify the *address_family*, which can change the interpretation of the remaining parameters. Currently, just the ''inet'' address family is supported.

For the Internet family, the address is either an Internet address expressed in the Internet standard ''dot notation'' (see *inet*(3N)), or a hostname present in the *hosts*(4) file, */etc/hosts*. (Other *hosts* databases, such as *named* and NIS, are ignored.)

Only the superuser can modify the configuration of a network interface.

The following parameters can be set with *ifconfig*:

up Mark an interface **up**. This can 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 is reinitialized.

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

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 Internet addresses and 10Mb/s Ethernet addresses.

–arp Disable the use of the Address Resolution Protocol.

alias *addr* Establish an additional network address for this interface. This can be useful in permitting a single physical interface to accept packets addressed to several different addresses such as when you are changing network numbers and you wish to accept packets addressed to the old interface. Another case is when you'd like to have multiple

addresses assigned to a single network interface. The **broadcast** and **netmask** options can also be used in conjunction with the **alias** option. When using aliases you may have to change the configuration of *routed*, especially if aliases are on different networks than the primary address. Aliases are added as host entries in the routing tables for *routed*. See *routed*(1M) for more information on this.

−alias|delete *addr*
Deletes a previously added alias.

metric *n* Set the routing metric of the interface to *n*, default 0. The routing metric is used by the routing protocol (*routed*). Higher metrics have the effect of making a route less favorable; metrics are counted as addition hops to the destination network or host.

netmask *mask* Specify how much of the address to reserve for subdividing networks into subnetworks. 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 *networks*(4). The mask contains 1's for the bit positions in the 32-bit address that 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.

broadcast *addr* 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.

dest_addr Specify the address of the correspondent on the other end of a point-to-point link.

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

−debug Disable driver-dependent debugging code.

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

NOTE

Network interfaces on Silicon Graphics systems can only receive and not send packets that use ''trailer'' link-level encapsulation. Therefore, *ifconfig* does not accept the **trailers** parameter.

DIAGNOSTICS

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

FILES

/etc/hosts	host-address database
/etc/config/ifconfig-*?*.options	site-specific options (1 file per interface)
/etc/config/ipaliases.options	interface-specific ip alias addresses

SEE ALSO

netstat(1), network(1M).

NAME

init, **telinit** – process control initialization

SYNOPSIS

/etc/init [**0123456SsQqabc**]
/etc/telinit [**0123456SsQqabc**]

DESCRIPTION

init

init is a general process spawner. Its primary role is to create processes from information stored in an **inittab** file (see **inittab**(4)). The default **inittab** file used is **/etc/inittab**; other files can be specified using the **INITTAB** keyword in the **system** file (see **system**(4)).

At any given time, the system is in one of eight possible run levels. A run level is a software configuration of the system under which only a selected group of processes exist. The processes spawned by **init** for each of these run levels is defined in **inittab**. **init** can be in one of eight run levels, **0**–**6** and **S** or **s** (run levels **S** and **s** are identical). The run level changes when a privileged user runs **/etc/init**. This user-level **init** sends appropriate signals to the original **init** (the one spawned by the operating system when the system was booted) designating the run level to which the latter should change.

The following are the arguments to **init**.

0 Shut the machine down so it is safe to remove the power. Have the machine remove power if it can.

1 Put the system into system administrator mode. All filesystems are mounted. Only a small set of essential kernel processes run. This mode is for administrative tasks such as installing optional utilities packages. All files are accessible and no users are logged in on the system.

2 Put the system into multi-user state. All multi-user environment terminal processes and daemons are spawned.

3 Start the remote file sharing processes and daemons. Mount and advertise remote resources. Run level **3** extends multi-user mode and is known as the remote-file-sharing state.

4 Define a configuration for an alternative multi-user environment. This state is not necessary for normal system operations; it is usually not used.

5 Stop the IRIX system and enter firmware mode.

6 Stop the IRIX system and reboot to the state defined by the **initdefault** entry in **inittab**.

a,**b**,**c** Process only those **inittab** entries for which the run level is set to **a**, **b**, or **c**. These are pseudo-states that can be defined to run certain commands but do not cause the current run level to change.

Q,**q** Re-examine **inittab**.

S,**s** Enter single-user mode. When the system changes to this state as the result of a command, the terminal from which the command was executed becomes the system console.

This is the only run level that doesn't require the existence of a properly formatted **inittab** file. If this file does not exist, then by default the only legal run level that **init** can enter is the single-user mode.

The set of filesystems mounted and the list of processes killed when a system enters system state **s** are not always the same; which filesystems are mounted and which processes are killed depends on the method used for putting the system into state **s** and the rules in force at your computer site. The following paragraphs describe state **s** in three circumstances: when the system is brought up to **s** with **init**; when the system is brought down (from another state) to **s** with **init**; and when the system is brought down to **s** with **shutdown**.

When the system is brought up to **s** with **init**, the only filesystem mounted is **/** (root). Filesystems for users' files are not mounted. With the commands available on the mounted filesystems, you can manipulate the filesystems or transition to other system states. Only essential kernel processes are kept running.

When the system is brought down to **s** with **init**, all mounted filesystems remain mounted and all processes started by **init** that should be running only in multi-user mode are killed. Because all login related processes are killed, users cannot access the system while it's in this state. In addition, any process for which the **utmp** file has an entry is killed. Other processes not started directly by **init** (such as **cron**) remain running.

When you change to **s** with **shutdown**, the system is restored to the state in which it was running when you first booted the machine and came up in single-user state, as described above. (The **powerdown**(1M) command takes the system through state **s** on the way to turning off the machine; thus you can't use this command to put the system in system state **s**.)

When an IRIX system is booted, **init** is invoked and the following occurs. First, **init** looks in **inittab** for the **initdefault** entry (see **inittab**(4)). If there is one, **init** usually uses the run level specified in that entry as the initial run level for the system. If there is no **initdefault** entry in **inittab**, **init** requests that the user enter a run level from the virtual system console. If an **S** or **s** is entered, **init** takes the system to single-user state. In the single-user state the virtual console terminal is assigned to the user's terminal and is opened for reading and writing. The command **/sbin/sulogin** is invoked, which prompts the user for a root password (see **sulogin**(1M)), and a message is generated on the physical console saying where the virtual console has been relocated. If **/bin/sulogin** cannot be

found, then **init** attempts to launch a shell: looking first for **/bin/csh**, then for **/sbin/sh**, then finally for **/bin/ksh**. 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** only re-initializes to the single-user state if the **inittab** file does not exist.

If a **0** through **6** is entered, **init** enters the corresponding run level. Run levels **0**, **5**, and **6** are reserved states for shutting the system down. Run levels **2**, **3**, and **4** are available as multi-user operating states.

If this is the first time since power up that **init** has entered a run level other than single-user state, **init** first scans **inittab** for **boot** and **bootwait** entries (see **inittab**(4)). These entries are performed before any other processing of **inittab** takes place, providing that the run level entered matches that of the entry. In this way any special initialization of the operating system, such as mounting filesystems, can take place before users are allowed onto the system. **init** then scans **inittab** and executes all other entries that are to be processed for that run level.

To spawn each process in **inittab**, **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 **inittab**, **init** waits for one of its descendant processes to die, a powerfail signal, or a signal from another **init** or **telinit** process to change the system's run level. When one of these conditions occurs, **init** re-examines **inittab**. New entries can be added to **inittab** at any time; however, **init** still waits for one of the above three conditions to occur before re-examining **inittab**. To get around this, the **init Q** (or **init q**) command wakes **init** to re-examine **inittab** immediately. Note, however, that if the **inittab** has been edited to change baud-rates, those changes only take effect when new **getty** processes are spawned to oversee those ports. Use **killall getty** to terminate all current **getty** processes, then **init Q** to re-examine the **inittab** and respawn them all again with the new baud-rates.

When **init** comes up at boot time and whenever the system changes from the single-user state to another run state, **init** sets the **ioctl**(2) states of the virtual console to those modes saved in the file **/etc/ioctl.syscon**. This file is written by **init** whenever the single-user state is entered.

When a run level change request is made **init** sends the warning signal (**SIGTERM**) to all processes that are undefined in the target run level. **init** waits five seconds before forcibly terminating these processes via the kill signal (**SIGKILL**).

When **init** receives a signal telling it that a process it spawned has died, it records the fact and the reason it died in **/var/adm/utmp** and **/var/adm/wtmp** if it exists (see **who**(1)). A history of the processes spawned is kept in **/var/adm/wtmp**.

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.

telinit

telinit, which is linked to **/sbin/init**, is used to direct the actions of **init**. It takes a one-character argument and signals **init** to take the appropriate action.

FILES

/etc/inittab default **inittab** file
/var/adm/utmp
/var/adm/wtmp
/etc/TIMEZONE
/etc/ioctl.syscon
/dev/console

SEE ALSO

getty(1M), killall(1M), login(1), powerdown(1M), sh(1), shutdown(1M), stty(1), sulogin(1M), who(1), kill(2), inittab(4), system(4), timezone(4), utmp(4), termio(7).

DIAGNOSTICS

If **init** finds that it is respawning an entry from the **inittab** file more than ten times in two minutes, it assumes that there is an error in the command string in the entry, and generates an error message on the system console. It then refuses to respawn this entry until either five minutes has elapsed or it receives a signal from a user-spawned **init** or **telinit**. This prevents **init** from consuming system resources when someone makes a typographical error in the **inittab** file or a program is removed that is referenced in **inittab**.

When attempting to boot the system, failure of **init** to prompt for a new run level may be because the virtual system console is linked to a device other than the physical system console.

NOTES

init and **telinit** can be run only by a privileged user.

The **S** or **s** state must not be used indiscriminately in the **inittab** file. A good rule to follow when modifying this file is to avoid adding this state to any line other than the **initdefault**.

If a default state is not specified in the **initdefault** entry in **inittab**, state **6** is entered. Consequently, the system loops; that is, it goes to firmware and reboots continuously.

If the **utmp** file cannot be created when booting the system, the system boots to state **s** regardless of the state specified in the **initdefault** entry in the **inittab** file.

In the event of a file table overflow condition, **init** uses a file descriptor associated with the **inittab** file that it retained from the last time it accessed that file. This prevents **init** from going into single user mode when it cannot obtain a file descriptor to open the **inittab** file.

The environment for **init** and all processes directly started by is set initially from a table that is builtin, and then by parsing the file **/etc/TIMEZONE**. Lines in that file that are too long are ignored.

NAME

inst – software installation tool

SYNOPSIS

inst [**−anAMNQXY**] [**−f** source] [**−m** hardware**=**value] [**−r** target]
[**−u** action] [**−F** selections-file] [**−I** selection]
[**−R** selection] [**−K** selection] [**−X** file] [**−Y** file]
[**−V** resource**:**value]

DESCRIPTION

inst is the installation tool used to install, upgrade, or remove software distributed by Silicon Graphics. There are two ways to run *inst*:

– Invoke *inst* as a command from the shell.

This is known as invoking *inst* using *IRIX Installation* and you must be superuser to do this. Some software cannot be installed using IRIX Installation (Release Notes and *inst* itself warn you about this software) and some commands within *inst* cannot be performed when using IRIX Installation. This is due to system integrity problems that can arise from changing some software or performing certain operations while that software is running.

– Invoke *inst* in a standalone mode.

To invoke *inst* in a standalone mode (known as *Miniroot Installation*), you shut down the system to the PROM monitor level (see *shutdown*(1M)), and load a collection of files known as the *miniroot* into the swap partition of your system disk. The miniroot contains a UNIX kernel, *inst* and several other programs. *inst* is automatically invoked after you load the miniroot and the **/** and **/usr** filesystems are automatically mounted as **/root** and **/root/usr**. New versions of IRIX and some software options must be installed from the miniroot. Directions for loading the miniroot are in the *IRIX Admin: Software Installation and Licensing* guide.

The software that you install using *inst* is known as a *software distribution*. Software distributions are prepared by Silicon Graphics and are in a format that can be read only by *inst*.

Software distributions can be on 1/4" cartridge tapes, on CD-ROM discs (CDs), and on disk. *inst* can read the distribution from a drive (tape, CD-ROM or disk) mounted on the same workstation that the software will be installed on (known as the *local* workstation), or from a tape drive, CD-ROM drive or disk on another workstation (known as the *remote* workstation) connected to the same network. If a distribution is on disk, it is in a directory called a *distribution directory*.

In order to install software from a distribution on a remote workstation (tape, CD-ROM, or distribution directory), the user ID that you use must have read permission for the device or distribution directory. By default, *inst* uses the current user ID and if the connection can't be established the user ID **guest** is used. For any user ID this requires that either there is no password for the account on the remote workstation or that the user ID has been added to the *.rhosts* file. See the example below. A different user ID can be

specified with the **–f** option (see below) or the **from** command within *inst*. The user ID for any account that does not have a password will work if it is able to read the distribution directory or device. If an account with an assigned password must be used, the *.rhosts* file for that user ID on the remote workstation must contain the name of the local workstation and the user ID. For example, the file */usr/people/joe/.rhosts* on 'bigserver' would contain the line:

```
rock.csd.sgi.com    joe
```

When **joe** wants to install C++ on 'rock' and the software distribution for C++ is located on bigserver in the directory */d/newrelease*, he would enter the command:

```
inst –f joe@bigserver:/d/newrelease/c++
```

The *.rhosts* file must have the correct ownership and permissions for access to be granted. See the *hosts.equiv*(4) reference page for details.

When using a distribution on a remote workstation, the file */usr/etc/inetd.conf* on the remote workstation and on any gateway workstations between the local and remote workstation needs to be modified. See the *IRIX Admin: Software Installation and Licensing* guide for details.

When *inst* first comes up, it displays the default location of the software distribution and possibly the user ID it is using. When using IRIX Installation, the default location of the software distribution will be the location of the software distribution that you last installed. The **–f** option (see below) can be used to specify the location of the the software distribution when IRIX Installation is used. This sets the default location that will be reported when *inst* comes up. In the case of a Miniroot Installation, the default location is wherever the miniroot came from. If the distribution is **none**, *inst* does not load one automatically; this is useful for removing or browsing just the installed software. Within *inst*, the **from** command can be used to change the distribution location.

The *inst* command line options are:

–a Execute *inst* from IRIX with no interaction from the user (automatic mode). No menus appear and the default location of the software distribution is used unless the **–f** option is given. The software that will be installed is selected by *inst* using an algorithm described in the *IRIX Admin: Software Installation and Licensing* guide.

–A A shorthand for **–a –u all**.

–f *source*
Specify the location of the software distribution. The format of *source* depends on the location of the distribution and other factors. The possibilities are:

tape	local tape drive
/dev/nrtape	local tape drive
/CDROM/dist	local CD-ROM drive
distdir	local distribution directory *distdir*
/CDROM/dist/*product*	installing just *product* from a local CD-ROM drive
*distdir***/***product*	installing just *product* from a local distribution directory
none	for browsing or removing installed software only
*server***:/dev/tape**	tape drive on a remote workstation whose hostname is *server*
server:*CDdir***/dist**	CD-ROM drive with the CD mounted at *CDdir* on a remote workstation whose hostname is *server*
server:*distdir*	distribution directory *distdir* on a remote workstation whose hostname is *server*
server:*CDdir***/dist/***product*	installing just *product* from a CD mounted at *CDdir* on a remote workstation whose hostname is *server*
server:*distdir*/*product*	installing just *product* from a distribution directory *distdir* on a remote workstation whose hostname is *server*

When using a remote distribution source, you can prepend *user***@** to specify a user ID other than guest as discussed above. For example, to use a remote tape drive on bigserver as user **instuser**, the **−f** argument is **instuser@bigserver:/dev/tape**.

−r *target*
Set the effective root directory in which the new software will be installed. By default, this is **/**. This option is used to install software somewhere other than the default location and to specify the root of a virtual IRIX tree for diskless prototype trees.

−n Initialize the *dryrun* resource to *true*. See RESOURCES.

−C Install software for all available architectures. This option is normally used only by *share_inst*(1M) when constructing a share-tree for diskless clients. When this option is given *inst* installs the sharable software from the specified distribution for as many different client architectures as possible.

–I *selection*
Specify products or subsystems to install, where *selection* is a product name, or other expression accepted by the *inst* **install** command. Multiple expressions can be specified by using a comma separated list or by using multiple **–I** options.

–R *selection*
Specify products or subsystems to remove, where *selection* is a product name, or other expression accepted by the *inst* **remove** command. Multiple expressions can be specified by using a comma separated list or by using multiple **–R** options.

–K *selection*
Specify products to keep (don't install or remove), where *selection* is a product name, or other expression accepted by the *inst* **keep** command. Multiple expressions can be specified by using a comma separated list or by using multiple **–K** options.

–V *resource***:***value*
Initialize *resource* to *value*. See RESOURCES.

–M Do not display *inst* output using its built-in pager, and do not show percent-done messages. This is the default when running an automatic install.

–E Invoke **inst_terse_mode**. Provides reduced output for scripting. Equivalent to setting **inst_terse_mode=true** in the shell environment before invoking *inst*. If set true, it minimizes *inst* output: most normal status and progress messages are suppressed unless explicitly asked for, but any warnings, errors, and explicitly requested output still appear. See the description of the preference **inst_terse_mode** within *inst* or *swmgr* for more information.

–F *selections-file*
The *selections-file* is used to pre-select subsystems for installation or removal. See SELECTIONS FILE.

–P *file* Specify *file* to be **precious**. See RESOURCES.

–N Disables space-checking by initializing the *space_check* resource to *false*. See RESOURCES.

–u *action*
Specify the type of installation action to use. The list of different actions are:

new Select all new products for installation.

upgrade Select all upgrade products for installation.

upgrade_and_new Select all new and upgrade products for installation.

upgrade_or_new Select upgrade (or new, if no upgrades) products for installation.

upgrade_and_rnds_or_new
Select upgrade and related new default subsystems (or new defaults, if no upgrades, no downgrades and no same) for installation. (This is the default action.)

all Select all products for installation.

–X *file*
Exclude *file* during installations and removals. If *file* is a directory, its descendents are also excluded. Multiple **-X** options are permitted, however wildcards are not. To quickly rebuild a corrupted installation history, re-install the same versions of your existing software using *-X/* (no files will be touched). See also the *exclusions* resource.

–Y *file*
Install only *file* during installations and removals. If *file* is a directory, only it and its descendents are installed or removed. Multiple **-Y** options are permitted, however wildcards are not. See the *restrictions* resource.

–U Run the tool in *extract_mode*, useful for recreating a source tree from a set of images produced by gendist(1M). See RESOURCES.

–T Run the tool in *symlink_mode*, useful for creating a tree of symbolic links, instead of regular files. See RESOURCES.

–Q Execute *inst* in a special mode in which only rqs information about each executable is updated into the installation history.

–m *hardware*=*value*
The software distribution for some software products contains several hardware-specific copies of the same file. By default *inst* installs the copy that is appropriate for the type of workstation you are installing on. The **–m** option is useful when you are installing software on a disk that will be transferred to a different type of workstation or when you are extracting files from a software distribution for a different type of workstation. In both these cases you would probably also use the **–r** option. The *inst* command **admin hardware** can be used to view the current hardware variables in effect.

Acceptable arguments are:

CPUBOARD=*cpu*
GFXBOARD=*gfxboard*
SUBGR=*subgr*

default

The **default** keyword substitutes all of the hardware values for the current system. Acceptable values of *cpu* are: IP4, IP5, IP6, IP7, IP9, IP12, IP17, IP19, IP20, IP21, IP22, IP25, IP26, and IP28. The output of *hinv*(1M) gives the value of *cpu*. The output of *hinv* also contains a line describing the type of graphics on the workstation. Use this table to determine the values of *gfxboard* and *subgr*:

hinv Output	*cpu*	*gfxboard*	*subgr*
`Graphics board: GR1*`	IP6, IP12	ECLIPSE	ECLIPSE
`Graphics board: GR2-*`	IP12, IP20	EXPRESS	EXPRESS
`Graphics board: GR2-*`	IP17	EXPRESS	IP17
`Graphics board: GR3-*`	IP22	EXPRESS	EXPRESS
`Graphics board: GU1-*`	IP17	EXPRESS	IP17
`Graphics board: GU1-*`	IP22	EXPRESS	EXPRESS
`Graphics board: GU1-*`	`and XL`IP22	NEWPRESS	NEWPRESS
`Graphics board: LG1`	IP12	LIGHT	LIGHT
`Graphics board: LG1`	IP17	LIGHT	IP17
`Graphics board: LG1`	IP20	LIGHT	LG1MC
`Graphics board: XL`	IP22	NEWPORT	NG1
`Graphics board: Indy *-bit`	IP22	NEWPORT	NG1
`Graphics board: VGR2`	IP6, IP12	ECLIPSE	ECLIPSE
`Graphics option installed`	IP4	CLOVER1	IP4G
`GT Graphics option installed`	IP4, IP5, IP7	CLOVER2	*cpu*GT
`GT Graphics option installed`	IP9	CLOVER2	IP7GT
`GT Graphics option installed`	IP17	CLOVER2	IP17
`VGX Graphics option installed`	IP7, IP9	STAPUFT	IP7GT
`VGX Graphics option installed`	IP17	STAPUFT	IP17
`VGXT Graphics option installed`	IP7	STAPUFT	SKYWR
`VGXT Graphics option installed`	IP17	STAPUFT	IP17SKY
`RealityEngine Graphics option installed`	IP7	VENICE	IP7
`RealityEngine Graphics option installed`	IP17	VENICE	IP17
`RealityEngine Graphics option installed`	IP19	VENICE	IP19
*	IP21	VENICE	IP21
*	IP25	VENICE	IP25
`Graphics board: RE3`	IP19	KONA	IP19
*	IP25	KONA	IP25
*	IP22	MGRAS	MGRAS

*	IP26	MGRAS	I2_64
*	IP28	MGRAS	I2_64
*	IP19	EXPRESS	IP19
*	IP21	EXPRESS	IP21
*	IP25	EXPRESS	IP25
*	IP26	EXPRESS	I2_64
*	IP28	EXPRESS	I2_64
No ''Graphics'' line	any	SERVER	SERVER

* output contains additional text that is unimportant

When the software distribution is a CD in a remote CD-ROM drive, the program *cdinstmgr*(1) must be run on the remote workstation. It does the mounting of the CD and guards against accidental ejection of the CD by other users. When the software distribution is a CD in a local CD-ROM drive, the mounting of the CD is done automatically by *inst*.

Unless the **–a** option is given, a menu of commands will be displayed when *inst* is invoked. This menu, the Inst Main Menu, contains all of the commands that most *inst* users ever need. The **Admin Menu** contains less-commonly used administrative commands. The **View Menu** is used to control the order and presentation of software packages, and disk space requirements displayed by the list and step commands. The **Interrupt/Error Menu**, is displayed automatically when certain error conditions are encountered during installation. These menus and their commands are described fully in the *IRIX Admin: Software Installation and Licensing* guide.

inst has an extensive online help system. Enter the command **help** at the *inst* prompt to see the list of help topics. To view the information on a particular topic, enter the command **help** *topic*. The topic **overview** provides information about what *inst* does and how to use it. This is a good starting point for new users.

The *IRIX Admin: Software Installation and Licensing* guide is the reference manual for *inst*. Installation issues for specific products are discussed in the Release Notes for the different products.

RESOURCES

A number of resources are available for customizing *inst* and *swmgr*. Resources are loaded from */var/inst/.swmgrrc*, */.swmgrrc*, *~/.swmgrrc*, and from the command line, with command-line definitions having the highest precedence.

The keywords in the following descriptions are:

inst-only used only by *inst*
swmgr-only used only by *swmgr*(1M)
permanent saved across sessions

transient	not saved across sessions
beginner	most commonly used
expert	for experts only
hidden	for internal use or for debugging only
readonly	cannot be modified by the user

abort_on_error

Abort automatic installation on errors
Type: boolean (inst-only permanent expert)
Default: true
Controls how errors are handled during automatic installations. If "true" (the default) the installation terminates immediately. If "false" the installation continues. In either case the errors are reported in /var/inst/INSTLOG.

all_architectures

Install all architectures
Type: boolean (transient readonly hidden)
Default: false
Controls whether files for all architectures are installed. This preference applies during diskless share-tree or client-tree installations, and can only be set with the -C command-line option. See client_inst(1).

always_confirm_quit

Confirm quit action
Type: boolean (permanent)
Default: false
Controls whether the user is always asked for confirmation of a quit command.

always_page_inst

Controls whether not explicitly requested
Type: boolean (inst-only transient expert)
Default: false
inst output, such as progress messages during actual installation, are paged or display non-stop.

auto_inst_new

Select new products in automatic mode
Type: boolean (swmgr-only permanent expert)
Default: false
Controls whether new products are selected for installation in automatic mode (swmgr only.)

auto_inst_upgrades
Select upgrade products in automatic mode
Type: boolean (swmgr-only permanent expert)
Default: true
Controls whether upgrade products are selected for installation in automatic mode (swmgr only.)

autoconfig_overhead
Autoconfig overhead, in bytes
Type: int (transient expert)
Default: 179200
Controls the amount of disk space reserved during the installation for temporary disk space required by autoconfig(1M) to build a new kernel. Under special circumstances when a debug kernel is being built, this value should be doubled. See also "kernel_size_32" and "kernel_size_64".

automatic
Automatic installation
Type: boolean (transient hidden)
Default: false
Controls whether the installation proceeds without user intervention.

autoselect
Automatic installation selections
Type: boolean (permanent)
Default: true
Controls whether inst does an initial selection of subsystems to be installed when it is first started and after every "from" command.

background
Background the application
Type: boolean (swmgr-only permanent expert)
Default: true
Controls whether swmgr is run in the foreground or in the background when invoked from a shell.

beep Beep when installation completes
Type: boolean (permanent)
Default: true
Controls the audible signaling at the end of the "go" command.

busy_check
Busy file space checking
Type: boolean (transient expert)
Default: true
Controls whether busy files (those which are currently accessed by executing processes) are considered during the pre-installation space check. If busy files are removed or overwritten during the installation, their disk blocks are not immediately reclaimed by the operating system. Closing other applications before a live-install may be required in order for the disk space requirements to be met. In the miniroot, the busy check is always skipped.

checkpoint_restart
Automatically restart install if true
Type: boolean (transient expert)
Default: false
Used for checkpoint restarting. Automatically restart installation if set true.

checkpoint_restart_mode
Override normal checkpoint restart mode
Type: string (transient hidden)
Default: NULL
Specifies the initial user interface mode used for checkpoint restarting. Brings up swmgr in a specific mode overriding value set by user in custom_startup_mode.

checkpoint_selections
Checkpoint before go as well as during go
Type: boolean (permanent expert)
Default: false
Controls whether or not to checkpoint the selections during the selection process. This is useful when making complex selections worth saving.

clearprompt
Clear the "more?" prompt
Type: boolean (inst-only permanent)
Default: true
Controls the way in which some prompts are displayed. Certain operations use "throw away" prompts that are usually cleared by backspacing, then overwriten with spaces. For terminals that can’t clear prompts in this way, it is more appropriate to just move to the next line. Set clearprompt "off" to use the simple case. It is "on" by default and saved from session to session.

cmd_installs
Command-line install selections
Type: string (transient hidden readonly)
Default: NULL
List of products to install supplied on the command line.

cmd_keeps
Command-line keep selections
Type: string (transient hidden readonly)
Default: NULL
List of products to keep supplied on the command line.

cmd_patch_hist_removes
Command-line remove selections
Type: string (transient hidden readonly)
Default: NULL
List of patch products to remove history supplied on the command line.

cmd_removes
Command-line remove selections
Type: string (transient hidden readonly)
Default: NULL
List of products to remove supplied on the command line.

columns
Number of columns in tty display
Type: int (inst-only permanent)
Default: 0
The "columns" variable is not supported in this release, but will be supported in a future release. In this release the number of columns is automatically detected by inst (if possible), but cannot be overridden.

confirm_nfs_installs
Confirm installs onto nfs-mounted filesystems
Type: boolean (permanent)
Default: on
Controls installs onto nfs-mounted filesystems. If set to "on", the user is asked to confirm any installations to nfs-mounted directories located on another host. When set to "off", inst or swmgr will install on to nfs-mounted filesystems as long as the user has the necessary permissions.

confirm_quit
Confirm quit if pending actions
Type: boolean (permanent)
Default: true
Controls whether the user is notified that install/remove actions are pending when quitting.

custom_startup_mode
Swmgr initial selections mode
Type: choice (permanent expert)
Choices: off,distribution,always
Default: false
Controls how swmgr is initially presented. Possible values are: off - always start in automatic mode; distribution - start in custom-selections mode if a distribution has been specified, otherwise in automatic mode; always - start in custom-selections mode if a distribution has been specified, otherwise in manage-installed-software mode.

debug_menu
Debug menu
Type: boolean (swmgr-only transient hidden)
Default: false
Controls whether the debug menu is available in the graphical tool.

default_config
Force default configuration files
Type: boolean (permanent expert)
Default: false
Controls whether to override the normal configuration file rules, and install ALL the configuration files from the distribution. If the user has modified these files, they are first saved with a ".O" suffix.

default_sharedirs
Default share directories
Type: string (transient readonly expert)
Default: /usr
Contains the list of default share directories used in a diskless share-tree or client-tree installation.

delay_conflicts
Controls when certain conflicts are presented
Type: choice (transient)
Choices: ask,on,off

Default: ask
Controls when certain "delayable" conflicts are presented to the user. Currently, only particular incompat conflicts may be delayed. The default behavior, "ask", will query the user when the first delayable conflict is presented. At that time, the user will have the option of resolving the conflict before the installation or prior to exiting. In some installation scenarios, the conflict will have to be delayed since the subsystem(s) necessary to resolve the conflict are not on the current distribution. After making a choice, this behavior will continue for the remainder of the installation session or until the preference value is explicitly changed. An "on" value will always present the delayable conflicts prior to exiting. An "off" selection will present all conflicts, including the delayable variety, prior to installation. This preference is not saved from session to session. To make this preference persistent, add "delay_conflicts :" and the desired value to the .swmgrrc file.

delay_exitops
Controls when exitops are executed
Type: boolean (permanent)
Default: false
Controls when exitops are executed. When this resource is false, the default case, exitops are executed immediately after all files are installed and before control is returned to the user. When this resource is true, the exitops will not be executed until the user quits. If the user interrupts an installation and chooses to save the exitops for later execution, any unexecuted exitops will be executed upon exiting the application.

delayspacecheck
Delay disk space checks until go
Type: boolean (permanent)
Default: false
Controls the timing of disk space calculations. If "delayspacecheck" is "off", disk space calculation is done when a "list", "step", or "space" command is given if no disk space calculation has yet been done for the current software distribution. If "delayspacecheck" is "on", disk space calculation is deferred until the "go" or "space" commands are given. When "delayspacecheck" is "on", no disk space information is displayed by the "list" or "step" commands.

detailspacecheck
Detailed space checking
Type: boolean (permanent)
Default: off
Controls the data used for disk space calculations. If "detailspacecheck" is "on", size and type of every file installed on the system via inst is obtained by stat-ing the file on the disk. If "detailspacecheck" is "off", the state of every file is assumed to be the same as that saved in the history database and that information is used to make the disk space calculations.

"detailspacecheck" is slower, but more accurate.

disable_keepfile
Suppress keepfile processing
Type: boolean (transient)
Default: false
If set, the normal keepfile processing is skipped. Entries in $rbase/var/inst/.keepfile will be ignored and the initial installation selections will be unaffected by the .keepfile entries. See the "keepfile" help topic.

disk_avail Override available space on target filesystems
Type: string (transient hidden)
Default: NULL
This preference specifies the free space (in bytes) used in disk space calculations. If set to a single value, like "50M" or "1000" then it will apply to all filesystems. To set the free space of specific filesystems, use the form: "fs:size|fs:size|..." For example, a value of "/:4096|/usr:50M" will result in 4096 bytes available on the / filesystem, and 50M on /usr. This resource is for debugging only. Also see the "disk_blocksize" and "disk_capacity" preferences.

disk_blocksize
Override blocksize of target filesystems
Type: string (transient hidden)
Default: NULL
This preference specifies an alternate blocksize (in bytes) used in disk space calculations. If set to a single value, like "1024", or "4k", then it will apply to all filesystems. To set the blocksize for specific filesystems, use the form: "fs:blocksize|fs:blocksize|..." For example, a value of "/:512|/usr:4k" will result in a blocksize of 512 bytes on the / filesystem, and 4096 byes on the /usr filesystem. Blocksizes should be a multiple of 512 bytes. This resource is for debugging only. Also see the "disk_avail" and "disk_capacity" preferences.

disk_capacity
Override capacity of target filesystems
Type: string (transient hidden)
Default: NULL
This preference specifies an alternate filesystem size (in bytes) used in disk space calculations. If set to a single value, like "50M" or "500k" then it will apply to all filesystems. To set the size of specific filesystems, use the form: "fs:size|fs:size|..." For example, a value of "/:300k|/usr:100M|/d2:100000" will result in a size of 300 kilobytes on the / filesystem, 100 megabytes on /usr, and 100000 bytes on /d2. This resource is for debugging only. Also see the "disk_avail" and "disk_blocksize" preferences.

diskless
Diskless mode
Type: choice (transient readonly hidden)
Choices: none,share,client
Default: none
Controls whether the tool is operating on a normal (non-diskless) tree, a diskless share tree, or a diskless client tree.

display_size
Units for product sizes
Type: choice (inst-only transient)
Choices: bytes,blocks,kbytes
Default: kbytes
Controls the units (bytes, kilobytes, or 512-byte blocks) used to display product sizes.

dist Most recent distribution source
Type: string (permanent)
Default: NULL
Each time product descriptions are read, the current software distribution source (see help from) becomes the value of "dist". (A product name, if included in the "from" argument, is not saved in "dist".) You can use the value of "dist" in your "from" commands using the syntax "$dist", for example "from $dist/eoe2". The command "from none" results in no distribution being read, and sets the view to target for browsing or removing installed software. The value of dist is saved from session to session.

distribution
The name of the distribution supplied by the user.
Type: string (transient hidden readonly)
Default: NULL
This is the name of the distribution supplied by the user on the command line. See also "dist".

dryrun
Dryrun mode
Type: boolean (transient hidden)
Default: false
Controls whether operating in dryrun mode. In this mode, no files are touched on the disk, and the names of files which would have otherwise been installed or removed are displayed.

error_coredump
Force coredump on X errors
Type: boolean (swmgr-only transient hidden)
Default: false
Controls whether errors in the X-Window interface result in an immediate coredump.

exclusions
List of excluded files
Type: string (transient expert)
Default: NULL
This resource holds the list of excluded files, separated by whitespace or the '|' character. Excluded files (and, for directories, their descendents) are not installed or removed during a "go". A value of "none" indicates no exclusions are in affect. The -X command line option may also be used to specify the exclusions. After modifying this preference with the "set" command, any previous disk space calculations may become out of date. Use the "admin recalc" command to determine the new space requirements. See also restrictions.

exitop_limit
Exitop output limit
Type: int (transient expert)
Default: 100000
This is the maximum output allowed from exitops (shell commands executed at the end of the installation). Output over this limit is not displayed or logged. Also applies to preops and postops (commands executed immediately before or after a file is installed) and removeops (executed after a file is removed). Removeops are only executed when a subsystem is removed, for example with "versions remove", but not when a subsystem is upgraded.

extract_mode
Extract images to original source tree
Type: boolean (transient readonly hidden)
Default: false
Controls whether inst or swmgr operates in extract mode. In this mode, files are installed under their original source names (relative to the effective root) used to create the images. Only regular files (not directories, symlinks, etc.) are installed. No conflict checking, rqs processing, or machtag matching is performed. This preference must be set before reading the distribution, preferably on the command-line using the -U option.

fullmenu
Display hidden commands
Type: boolean (inst-only permanent)
Default: false
Controls the list of commands shown in menus. If fullmenu is "off", (the default), the most commonly used commands are displayed in the menus. If fullmenu is "on", all commands that are available at the prompt are shown in the menu, and all the hidden commands are displayed, too. This option changes only the display of commands, not their availability. It is "off" by default and saved from session to session.

hide_image_products
Hides the image level products when set to true.
Type: boolean (permanent expert)
Default: true
Hides the image level products from view when set to true.

http_cache
Cache http files locally.
Type: boolean (permanent expert)
Default: false
When reading an http distribution over a slow network link, it may be safer to read each file as quickly as possible and cache it on local disk rather than keeping the network connection open for the entire course of the install. Setting http_cache ON turns on local caching.

http_cache_tmp_dir
The tmp dir where http cached files will be created.
Type: string (permanent expert)
Choices: /var/tmp
Default: /var/tmp
If http caching is on (see http_cache), this is the tmp directory where the cached files will be stored. If http caching is off, this preference is ignored.

http_picky
Require special format distribution file in an http distribution.
Type: boolean (permanent expert)
Default: true
Require special format distribution file in an http distribution, rather than trying to adapt to any directory format the http server may see fit to provide.

inst_initial_level
Inst product display level
Type: choice (permanent expert)
Choices: product,subsystem
Default: subsystem
Controls the initial level of products displayed in inst.

inst_terse_mode
Reduced output for scripting
Type: boolean (inst-only transient expert)
Default: false
If set true minimizes inst output: most normal status, and progress messages are suppressed unless explicitly asked for, but any warnings, errors and explicitly requested output still appear. Also forces page_output off, show_diskspace off, show_legend off, show_percent_done off, show_files off and verbosity to the value 1. Useful for automated scripts driving inst. See also the inst -F<selections-file> option, the inst "admin save" and "admin load" commands, and the verbose preference.

inst_visible_resources
Set of options to display.
Type: multivalue (inst-only permanent beginner)
Choices: permanent,transient,beginner,expert,tty,gui
Default: permanent,transient,tty
Controls the set of preferences displayed with the set command.

install_identical_files
Install files even if contents are not new
Type: boolean (permanent expert)
Default: true
Controls whether to extract files from the distribution even if the version on disk has the same size and checksum. If this resource is set to "false" installations are much faster when little has changed from release to release. Caution: use this option only if you are comfortable with the accuracy of the checksum -r test to determine whether two files of equal length indeed are identical. See sum(1m).

install_sites
List of former install sites
Type: string (permanent expert)
Default: NULL
Controls the list of former installation sites displayed.

install_sites_size
Number of items to keep in the install sites list.
Type: int (permanent)
Default: 10
Controls the number of items in the former installation sites list.

instmode
Type of installation
Type: choice (transient expert)
Choices: normal,prototype,client
Default: prototype
Controls certain details of how the installation is performed. The value is set automatically when the installation target is initialized. The instmode preference is exported as the environment variable $instmode for use by exitops. During installs into a bootable target (miniroot installs into /root, and live installs into /) instmode has the value "normal". During other installs the value is "prototype". If the value of "instmode" is changed, an "admin recalc" is also recommended to recalculate space requirements (if any) for the new UNIX kernel. Also see the "diskless" preference.

interactive
User interaction control
Type: boolean (transient hidden)
Default: true
Controls whether the user will be prompted for responses to questions.

kernel_size_32
Size of 32-bit kernel, in bytes
Type: int (transient expert)
Default: 3932160
Controls the amount of disk space reserved on 32-bit systems for /unix, during installations which will cause a new kernel to be built by autoconfig(1M). See also "kernel_size_64" and "autoconfig_overhead".

kernel_size_64
Size of 64-bit kernel, in bytes
Type: int (transient expert)
Default: 5767168
Controls the amount of disk space reserved on 64-bit systems for /unix, during installations which will cause a new kernel to be built by autoconfig(1M). See also "kernel_size_32" and "autoconfig_overhead".

lines Number of lines in tty display
Type: int (inst-only permanent)
Default: 0
Sorry, the "lines" variable is not supported in this release, but will be supported in a future release. In this release the number of lines is automatically detected by inst (if possible), but cannot be overridden.

log_pane_height
Height of the log pane in swmgr
Type: int (swmgr-only transient hidden)
Default: 125
Specifies the height of the log pane in swmgr.

log_pane_log_size
Size of the log pane display
Type: int (swmgr-only permanent hidden)
Default: 100000
Specifies the maximum amount of text that is displayed in the log pane before truncation occurs.

logfile_size
Size of log file in bytes before recycle
Type: int (permanent hidden)
Default: 102400
Controls the maximum size of the log file (/var/inst/INSTLOG).

mach_classfile
Mach classes file
Type: string (transient hidden readonly)
Default: /var/boot/.dl_classes
Holds the name of the file containing the hardware chart used during diskless installations when the -C argument is specified, requesting that all architectures be installed.

mach_info
Mach tag values
Type: string (transient hidden)
Default: NULL
Contains the mach tag values that determine which versions of hardware-specific files to install. Multiple mach expressions separated by the '|' character are permitted. If this preference is NULL (the default) the mach values are automatically derived according to the current hardware

configuration on the system.

machfile
Name of the standard mach file.
Type: string (transient hidden)
Default: /var/inst/machfile
Controls the location of the standard machfile.

menus
Print menus
Type: boolean (inst-only permanent)
Default: on
Controls the automatic display of menus. Once you are familiar with the menus, you may wish to disable the automatic display by setting this option "off". You can redisplay the current menu at any time with a "?" command. It is "on" by default and saved from session to session.

miniroot
Miniroot install
Type: boolean (transient hidden readonly)
Default: false
This flag is on if in the miniroot.

mock_sproc
Disable sprocs
Type: boolean (transient hidden)
Default: false
When set, this disables sproc'ing, so that debugging is possible. The command is printed on stdout, and execution terminates until something is read from stdin.

network_retry
Number of network retries
Type: int (permanent hidden)
Default: 2
Holds the number of successive network timeouts allowed before giving up on the connection. See timeout.

network_seek_threshold
Controls behavior of remote seek
Type: int (transient hidden)
Default: 65536
If seeking forward more than network_seek_threshold bytes in a remote file, re-start the dd process instead of reading and discarding the intervening bytes. This optimization of re-starting dd may be disabled by setting network_seek_threshold to zero. If the remote dd does not support the iseek option, network_seek_threshold is always treated as zero.

never_resize_pane
Do not resize panes when switching selections modes
Type: boolean (swmgr-only permanent expert)
Default: true
Controls whether to resize panes when switching between automatic installation and custom selections or manage installed software modes in swmgr.

neweroverride
Allows installation of older products
Type: boolean (transient)
Default: false
Controls whether older subsystems can replace newer subsystems that are installed. The default value, "off", prevents you from installing subsystems that are older than what is already installed (i.e. are marked "O" in "list" and "step" output). If you want to replace a subsystem with an older version, set "neweroverride" to "on". This option is not saved from inst session to inst session.

non_root
Not root user can perform installs/removals
Type: boolean (transient hidden readonly)
Default: false
Controls if a non-root user can perform installs/removals.

overprint
Use overprinting for verbose lists
Type: boolean (inst-only permanent)
Default: true
Controls the manner in which filenames are displayed during installations and removals if the preference show_files is turned on. When this option is on, scrolling the display is significantly reduced by overprinting the file names on the same line. If you prefer scrolling as the file names are displayed, set this option "off". It is "on" by default and saved from session to session.

override_space_check
Override space checking
Type: boolean (transient expert)
Default: false
If set to true, allows "go" despite a disk overflow condition.

page_output
Page output
Type: boolean (inst-only transient)
Default: true
Controls whether tty output is managed by a paging mechanism similar to more(1m).

perm_check
Permissions check
Type: boolean (transient hidden)
Default: true
Controls whether permissions-checking is performed during the pre-installation check. When this option is true, the inst or swmgr will inform you during the pre-installation check whether you have permission to install or remove files in the affected directories.

post_install_dialog
Ask about quit after install
Type: boolean (permanent expert)
Default: true
Controls if a dialog is displayed after an install asking if the user wants to quit.

precious_files
List of precious files
Type: string (transient expert)
Default: NULL
Holds the list of precious file separated by whitespace, specified on the cmd-line or .swmgrrc. Inst or swmgr will not overwrite or remove any files which the user has designated as precious.

promptforid
Interactive prompt for unknown uid
Type: boolean (permanent)
Default: false
Controls whether the user will be interactively queried for ids corresponding to unknown uids/gids. Unknown uids or gids result when a passwd or group file does not contain an id for the

given name. If "off", inst or swmgr will automatically choose the uids/gids (and write them to /usr/adm/SYSLOG) based on the startid variable.

remote_read_size
Controls the size of the buffer used for reading remote
Type: int (transient hidden)
Default: 10240
distributions. Controls the size of the buffer used for reading remote distributions.

report_exit_status
Set process exit status
Type: boolean (permanent hidden)
Default: false
Controls the exit status to be set to various non-zero values, depending on what caused the exit.

restricted
Restricted mod
Type: boolean (inst-only transient hidden)
Default: false
This resource controls restricted mode in inst.

restrictions
List of restricted files
Type: string (transient expert)
Default: none
Holds the list of restricted files, separated by whitespace or the '|' character. Restricted files (and, for directories, their descendents) are the only files installed or removed during a "go". A value of "none" indicates no restrictions are in affect. The -Y command line option may be used to specify the restrictions. After modifying this preference with the "set" command, any previous disk space calculations may become out of date. Use the "admin recalc" command to determine the new space requirements. See also exclusions.

rqs_only
Requickstart mode
Type: boolean (transient hidden)
Default: false
Controls if the only operation performed is rqs Controls if the only operation performed is rqs

rules_disable_defaults
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rules_disable_emptyTryAll
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rules_disable_emptyTryDefaults
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rules_disable_replaces
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rules_enable_allkids
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rules_include_rejected_defaults
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rules_nonbootable_ok
Rules debug
Type: boolean (transient hidden)
Default: false
Controls rules debugging

rulesoverride
Override any conflict
Type: boolean (transient)
Default: false
Controls whether conflicts can be overridden. The default value, "off", prevents you from installing subsystems that do not meet specified incompat or prereq rules. If you want to override these rules, set "rulesoverride" to "on". This option should be used cautiously. "rulesoverride" is not saved from session to session.

selections_pane_height
Height of the selections pane
Type: int (swmgr-only transient hidden)
Default: 350
Specifies the height of the selections pane.

set_path
Control $PATH variable for exitops
Type: choice (permanent expert)
Choices: default,environment,specific
Default: default
Can be set to default, environment, or specific. default: set path to a known path. environment: pass $path in from the environment $PATH (or use the default path if $PATH is not set) specific: use whatever is in the preference set_path_specific

set_path_specific
Path used for exitops
Type: string (permanent expert)
Default: NULL
Controls the path used for exitops.

shadow_files
List of files to shadow
Type: string (transient hidden)
Default: NULL
Holds the list of shadow files, separated by whitespace, specified on the command line or .swmgrrc

shadowing
File shadowing
Type: boolean (transient hidden readonly)
Default: NULL
Controls whether special files are shadowed during a live (non-miniroot) install.

sharebase
Root of sharetree in client mode
Type: string (transient hidden)
Default: NULL
Holds the name of the root of sharetree in client mode.

shell Program to use for "sh" and "shroot"
Type: string (permanent)
Default: /bin/csh
Controls what program to use for "sh" and "shroot". It may be any pathname, and is the name of the IRIX command that is invoked for "sh" and "shroot" commands. It is normally /bin/sh or /bin/csh; the default value comes from the SHELL environment variable. It is saved from session to session.

short_names
Display product name instead of description.
Type: boolean (permanent)
Default: false
Controls if the product name is displayed instead of the product description in swmgr.

show_absolute_sizes
Display absolute product sizes
Type: boolean (inst-only transient hidden)
Default: false
Controls the type of information shown in the size column in the output of the "list" and "step" commands. If this variable is set to "on", the total product size is shown. Otherwise the net change in product size (after the installation and removals) is shown.

show_command_pane
Display the command pane on startup
Type: boolean (swmgr-only transient hidden)
Default: false
Controls whether the command pane is shown on startup.

show_diskspace
Display diskspace summary
Type: boolean (inst-only permanent)
Default: true
Controls whether the diskspace is displayed by the list, step and recalculate commands. Use the "admin space" command to display this summary even when this preference is set to false. See also the preference inst_terse_mode.

show_distribution_pane
Display the distribution pane on startup
Type: boolean (swmgr-only transient hidden)
Default: true
Controls whether the distribution pane is shown on startup.

show_existing_conflicts
Display pre-existing conflicts
Type: boolean (transient hidden)
Default: false
Controls whether pre-existing conflicts are displayed or not.

show_files
Display filenames during install
Type: boolean (permanent expert)
Default: false
Controls whether the name of each file is displayed as it is installed or removed. See also the preference overprint to control whether these listed filenames are displayed on the same line or not. See also the preference verbose.

show_hidden_resources
Hide or present all preferences
Type: boolean (permanent expert)
Default: false
Controls whether the user is presented with the expert preferences, in addition to the basic

preferences.

show_legend
Display list legend
Type: boolean (inst-only permanent)
Default: true
Controls whether the legend is displayed at the beginning of the output of the list command. See also the preference inst_terse_mode.

show_lint
Show distribution consistency errors
Type: boolean (permanent expert)
Default: false
Controls whether to display any inconsistencies found while reading the distribution.

show_log_pane
Display the log pane on startup
Type: boolean (swmgr-only transient hidden)
Default: false
Controls whether the log pane is displayed in swmgr on startup.

show_percent_done
Display task percentages
Type: boolean (inst-only transient expert)
Default: true
Controls whether percent-done messages are displayed in inst. See also the preference verbose.

show_selections_pane
Display the selections pane on startup
Type: boolean (swmgr-only transient hidden)
Default: true
Controls whether the selections pane is shown on startup.

show_stat_pane
Display the status pane on startup
Type: boolean (swmgr-only transient hidden)
Default: true
Controls whether the status pane is shown on startup.

show_subtasks
Print tasking information
Type: boolean (transient hidden)
Default: false
Controls whether detailed information about tasks and sub-tasks is displayed.

skip_rqs
Skip the requickstart process entirely
Type: boolean (transient hidden)
Default: false
Controls whether to skip the requickstart process. Use this cautiously, because non-requickstarted executables will run much slower.

space_check
Controls space checking
Type: boolean (transient)
Default: true
Controls whether disk space checking is performed Use cautiously, or you can run out of diskspace.

space_indicator
Type of space indicator
Type: choice (swmgr-only permanent)
Choices: pie,bar,text
Default: pie
Controls the display of the space area, whether bar, pie or text.

space_update_interval
Seconds between disk space updates
Type: int (swmgr-only permanent)
Default: 10
Specifies how often the Disk Space area is automatically updated, (in addition to updates that occur when the product selections change.) If zero is specified, automatic updates are disabled.

startid
Initial id to assign for an unknown uid/gid
Type: int (permanent)
Default: 60000
This variable is only relevant if promptforid = "off". Inst or swmgr will automatically assign unknown uid/gid values based on the value of startid. After a given unknown uid or gid is

assigned the value startid, the next unknown uid/gid will be assigned startid-1, and the next startid-2, etc. It is recommended that startid be assigned a value that no current uids/gids possess. If inst or swmgr automatically assigns a uid or gid to a file, it will write the mapping to /usr/adm/SYSLOG.

startup_script

Controls how an optional installation startup script is handled
Type: choice (transient, expert,)
Choices: prompt,ignore,execute
Default: prompt
This resource is used to control how inst and SoftwareManager treat the inst.init startup script (if any) that accompanies the software distribution. If set to "prompt", the user is prompted before the script is executed. If set to "ignore", the startup script is ignored. If set to "execute" the startup script is run without confirmation from the user. Non-interactive installations (inst -a or swmgr -a) will fail if a startup script is present, unless this resource is explicitly set on the command-line with either "-Vstartup_script:ignore" or "-Vstartup_script:execute". This resource is not saved across sessions, unless a line such as "startup_script : ignore" is manually added to the /var/inst/.swmgrrc file. For security reasons you are strongly advised against adding the line "startup_script : execute" to the .swmgrrc file.

swmgr_initial_level

Swmgr product display level
Type: choice (swmgr-only permanent expert)
Choices: product,subsystem
Default: product
Controls the initial level of products displayed in swmgr.

swmgr_visible_resources

Set of optional preferences to display.
Type: multivalue (swmgr-only permanent beginner)
Choices: none,transient,expert
Default: none
Controls the set of optional preferences displayed in the preferences dialog.

swmgrrc_path

Search path for .swmgrrc preference files
Type: string (transient expert)
Default: /var/inst:$rbase/var/inst:$rbase$HOME:$HOME
Contains a colon-separated list of directories to search for preference files, such as ~/.swmgrrc, that contain preferences to be saved between sessions. The directories are searched in the order specified.

Preference settings in the last .swmgrrc found have the highest precedence.

Whenever you use the "set" command to change the value of a "permanent" preference, its value will be saved in .swmgrrc (the one most recently loaded) and will remain set in future inst or swmgr sessions.

The swmgrrc_path can also be modified by setting the environment variable SWMGRRC_PATH. If the value is "none", then no .swmgrrc files will be read, or saved.

symlink_mode
Create a symlink tree
Type: boolean (transient readonly hidden)
Default: false
Controls whether inst or swmgr operates in symlink mode. In this mode, all files (but not directories, config files, or noshare files) are installed as absolute symbolic links pointing to the corresponding file in the real tree. This preference can only be set on the command line, using -T<symlink_root>. See symlink_root.

symlink_root
Root of real tree in symlink_mode
Type: string (transient readonly hidden)
Default: NULL
Holds the root of the real tree that symlinks point to, This preference can only be set on the command line, using -T<symlink_root>. See symlink_mode.

target Location of target tree
Type: string (transient hidden readonly)
Default: /
Holds the name of the root directory on the local filesystem where files are installed or removed. Swmgr and inst do not install or remove files outside of the target. Normally during a live installation, the target is /, and in the miniroot it is /root. The -r command line option may be used to specify an alternate target.

timeout
Network timeout in seconds
Type: int (permanent)
Default: 120
Holds the amount of time to wait for a reply from a remote host. If the remote machine does not respond after this time, the request is attempted again. See network_retry. The default is two minutes; if your network is particularly slow, this may not be enough. If you receive errors

regarding the network timeout, and you believe the access will succeed if given more time, increase the value of this option. The value of timeout is saved from session to session.

toolname
Name of invoked installation program.
Type: string (transient hidden)
Default: NULL
Holds the name of the installation program (inst or swmgr).

trace Trace debugging
Type: multivalue (transient hidden)
Choices:
none,category,checkpoint,config,conflict,constructor,destructor,diskless,dispatcher,errno,file,general,gui,hist,ht
Default: NULL
Controls which type of debugging information is displayed. Can be "all" or any combination of categories. Set to "none" to disable debugging.

use_last_dist
Use last distribution
Type: boolean (swmgr-only permanent expert)
Default: false
Use the last distribution specified if none is specified this session. If no distribution was specified for this session, automatically load the last distribution seen (swmgr only.)

verbose
Display more detailed output
Type: boolean (permanent expert)
Default: false
When set true, causes more status messages to be displayed. For finer control of the same facility, see the verbosity setting. See also the preferences show_files, show_percent_done and inst_terse_mode.

verbosity
Message verbosity display threshold
Type: int (permanent)
Choices: 0 to 6
Default: 2
Controls the level of error and related info messages displayed. 0 for silent, 2 for verbose off (default), 4 for verbose on, 6 for all messages. For another way to control this setting, see the

verbose preference.

verify_checksum
Verify checksum of file during install
Type: boolean (permanent expert)
Default: on
During the installation, verify the checksum of each file as it is uncompressed or read from the archive.

wrapmode
Wrap/truncate long lines in display
Type: choice (inst-only permanent)
Choices: wrap,truncate
Default: wrap
Controls the display of long subsystem descriptions by the "list" command. In wrap mode, the part of the description that overflows the right margin of the screen is shown indented on the next line. In truncate mode, the line is truncated near the right margin. The "step" command always uses truncate mode, regardless of the setting of this option. It is "wrap" by default and saved from session to session.

SELECTIONS FILE

A **selections-file** can be used to specify the distribution and the selections for a single installation session. Blank lines and comment lines that begin with a pound-sign (**#**) are ignored. All other lines must be selections file directives.

The software distribution is specified using the **from** directive. This directive can be omitted from the selections file if the distribution has already been read and the user uses the **admin load** command. A selections file can contain only one distribution directive and it can appear anywhere in the selections file.

from *distribution*
from /CDROM/dist
from *host***:***distribution*

inst resource values can be specified for the installation session by using the **set** directive. These directives can appear anywhere in the file and all resource directives are interpreted together in the order that they appear.

set *resource value*

The product/subsystem selections for the installation session are specified using the selections directives. The selections directives can appear anywhere in the file and all selections directives are interpreted

together in the order that they appear. The directives are specified using their abbreviated forms, and include **install** (**i**), **remove** (**r**), **keep** (**k**), **don't install** (**di**), **don't remove** (**dr**). The directives **k**, **di**, and **dr** are semantically equivalent, and they all unmark the product(s) for either installation or removal. In the examples below, *prod_spec*, can be a product (*prod*), image (*prod.image*), or subsystem (*prod.image.subsys*).

i *prod_spec prod_spec ...*
r *prod_spec prod_spec ...*
k *prod_spec prod_spec ...*
di *prod_spec prod_spec ...*
dr *prod_spec prod_spec ...*

The *prod_spec* can also be a product status specifier when used with the install or keep directive. The product status specifier **new** causes the install or keep command to only select products that are not already installed. The product status specifier **upgrade** causes the install or keep command to only select products that are upgrades of installed products.

KEEP FILE

A **keepfile** can be used to prevent unwanted products from being marked for default installation. If the file */var/inst/.keepfile* exists, its contents are processed each time a new distribution is loaded. The keepfile lists new (N) products, images, or subsystems that are not to be marked for default installation. Upgrade (U) subsystems are not affected by the keepfile. The keepfile only affects the initial selections made by inst or swmgr when a distribution is loaded.

The *inst* command **admin updatekeepfile** updates the keepfile with respect to the current selections. New default subsystems that are not selected for installation are appended to the keepfile.

The keepfile contains one pattern per line. Only the first whitespace-separated pattern on each line is considered. A wildcard character, *, can be used in the pattern. For example eoe.books.* turns off default installation of all new subsystems in the eoe.books image. Comments beginning with # and ending with a newline are ignored.

FILES

/var/inst/.swmgrrc	most recent values of **set** options
/var/inst/.keepfile	new subsystems not selected for default install
/var/inst/help1	source for online help
/var/inst/hist	binary file that contains the installation history of your workstation
/var/inst/*product*	binary files, one per product available in any distribution (or since last time filesystems were remade)

SEE ALSO

distcp(1M), showfiles(1M), showprods(1M), swmgr(1M), versions(1M).

IRIX Admin: Software Installation and Licensing

NAME

killall – kill named processes

SYNOPSIS

killall [[–]signal]
killall [**–gv**] [**–k** secs] [[–]signal] [pname ...]
killall [**–gv**] [**–k** secs] [–signame] [pname ...]
killall –l

DESCRIPTION

killall sends a signal to a set of processes specified by name, process group, or process ID. It is similar to *kill*(1), except that it allows processes to be specified by name and has special options used by *shutdown*(1M).

When no processes are specified, *killall* terminates all processes that are not in the same process group as the caller. This form is for use in shutting down the system and is only available to the superuser.

The options to *killall* are:

signal, *–signal*
: Specifies the signal number. The minus (–) is required if *pname* looks like a signal number. If no signal value is specified, a default of **9** (KILL) is used.

pname
: When a process is specified with *pname*, *killall* sends *signal* to all processes matching that name. This form is available to all users provided that their user ID matches the real, saved, or effective user ID of the receiving process. The signal number must be preceded by a minus (–) if *pname* looks like a signal number.

signame
: A mnemonic name for the signal can be used; see the **–l** option.

–g
: Causes the signal to be sent to the named processes' entire process group. In this form, the signal number should be preceded by – in order to disambiguate it from a process name.

–k *secs*
: Allows the user to specify a maximum time to die for a process. With this option, an argument specifying the maximum number of seconds to wait for a process to die is given. If after delivery of the specified signal (which defaults to SIGTERM when using the **–k** option), *killall* waits for either the process to die or for the time specified by *secs* to elapse. If the process does not die in the allotted time, the process is sent SIGKILL.

–l
: Lists the signal names (see *kill*(1) for more information about signal naming). For example,

```
killall 16 myproc
killall -16 myproc
killall -USR1 myproc
```

are equivalent.

–v Reports if the signal was successfully sent.

killall can be quite useful for killing a process without knowing its process ID. It can be used to stop a runaway user program without having to wait for *ps*(1) to find its process ID. It can be particularly useful in scripts, because it makes it unnecessary to run the output of *ps*(1) through *grep*(1) and then through *sed*(1) or *awk*(1).

FILES

/etc/shutdown

SEE ALSO

fuser(1M), kill(1), ps(1), shutdown(1M), signal(2).

NAME

lboot – configure bootable kernel

SYNOPSIS

lboot *options*

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. System files in the directory *system* are used by *lboot* to determine which modules are to be configured into the kernel.

If a module in *master* is specified in the *system* file via "INCLUDE:", that module is 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 specified size. If the read succeeds, the device is assumed to exist, and its module is also included in the bootable kernel.

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.

Master files that are specified in the *system* file via "USE:" are treated as though the file were specified via the "INCLUDE:" directive, if an object file corresponding to the master file is found in the *boot* directory. If no such object file is found, "USE:" is treated as "EXCLUDE:".

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. Unless directed otherwise in the *system* file, the information is compiled with **$TOOLROOT/usr/bin/cc** and combined with the modules in the *boot* directory using **$TOOLROOT/usr/bin/ld**.

The options are:

–m *master* Specifies the directory containing the master files to be used for the bootable kernel. The default *master* directory is *$ROOT/var/sysgen/master.d*.

–s *system* Specifies the directory containing the system files. The default *system* directory is *$ROOT/var/sysgen/system*.

–b *boot* Specifies the directory where object files are to be found. The default *boot* directory is *$ROOT/var/sysgen/boot*.

–n *mtune* Specifies the directory where tunable parameters are to be found. The default *mtune* directory is *$ROOT/var/sysgen/mtune*.

–c *stune* Specifies the name of the file defining customized tunable parameter values. The default *stune* file is *$ROOT/var/sysgen/stune*.

–r *ROOT* **ROOT** becomes the starting pathname when finding files of interest to *lboot*. Note that this option sets **ROOT** as the search path for include files used to generate the target kernel. If this option is not specified, the **ROOT** environment variable (if any) is used instead.

–v Makes *lboot* slightly more verbose.

–u *unix* 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 Displays debugging information about the devices and modules put in the kernel.

–a Used to auto-register all dynamically loadable loadable kernel modules that contain a **d** and an **R** in their master files. Only the auto-register is performed, a kernel is not configured.

–A Disables auto-registering of all dynamically loadable modules. A kernel is produced, but no auto-registration is performed.

–l Used to ignore the **d** in all master files and link all necessary modules into the kernel.

–e Causes the result of whether an auto-config would have been performed to be printed, but no actual configuration is built.

–w Used to specify a work directory into which the *master.c* and *edt.list* files are written. By default these files are written into the *boot* directory.

–t 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* files, the object files in the *boot* directory, the modification time of the boot directory, the configuration files in the *master.d* directory and the modification time of the *stune* file with that of the existing 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

–T Performs the same function as the **–t** option, but does not prompt you to proceed.

–O *tags* Specifies tags to be used to select which tunable parameters to use as part of the the kernel build. Multiple **–O** options may be given.

–L *master* Specifies the name of the dynamically loadable kernel module to load into the running kernel. *master* is the name of a master file in the *$ROOT/var/sysgen/master.d* directory.

–R *master* Specifies the name of the dynamically loadable kernel module to register. *master* is the name of a master file in the *$ROOT/var/sysgen/master.d* directory.

–U *id* Used to unload a dynamically loadable kernel module. *id* is found by using the *lboot –V* command.

–W *id* Used to unregister a dynamically loadable kernel module. *id* is found by using the *lboot –V* command.

–V Used to list all of the currently registered and loaded dynamically loadable kernel modules.

It is best to reconfigure the kernel on a system with the *autoconfig* command.

EXAMPLE

`lboot -s` *newsystem*

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

FILES

/var/sysgen/system
/var/sysgen/master.d/*
/var/sysgen/boot/*
/var/sysgen/mtune/*
/var/sysgen/stune

SEE ALSO

autoconfig(1M), setsym(1M), systune(1M), master(4), mload(4), mtune(4), stune(4), system(4).

NAME

login – sign on

SYNOPSIS

login [**-d** *device*] [*name* [*environ* ...]]

DESCRIPTION

The **login** command is used at the beginning of each terminal session and allows you to identify yourself to the system. It is invoked by the system when a connection is first established. It is invoked by the system when a previous user has terminated the initial shell by typing a <Ctrl-d> to indicate an end-of-file.

If **login** is invoked as a command, it must replace the initial command interpreter. This is accomplished by typing

```
exec login
```

from the initial shell.

login asks for your user name (if it is not supplied as an argument) and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it does not appear on the written record of the session.

login reads **/etc/default/login** to determine default behavior. To change the defaults, the system administrator should edit this file. The examples shown below are login defaults. Recognized values are:

CONSOLE=*device* If defined, only allows root logins on the device specified, typically **/dev/console**. This MUST NOT be defined as either **/dev/syscon** or **/dev/systty**. If undefined, root can log in on any device.

PASSREQ=NO Determines whether all accounts must have passwords. If **YES**, and user has no password, they are prompted for one at login time.

MANDPASS=NO Like **PASSREQ**, but doesn't allow users with no password to log in.

ALTSHELL=YES If **YES**, the environment variable SHELL is initialized.

UMASK=022 Default umask, in octal.

TIMEOUT=60 Exit login after this many seconds of inactivity (maximum 900, or 15 minutes)

SLEEPTIME=1 Sleep for this many seconds before issuing "login incorrect" message (maximum 60 seconds).

DISABLETIME=20
After **LOGFAILURES** or **MAXTRYS** unsuccessful attempts, sleep for **DISABLETIME** seconds before exiting (no maximum).

MAXTRYS=3 Exit login after **MAXTRYS** unsuccessful attempts (0 = unlimited attempts).

LOGFAILURES=3 If there are **LOGFAILURES** consecutive unsuccessful login attempts, each of them is logged in **/var/adm/loginlog**, if it exists. **LOGFAILURES** has a maximum value of 20.

Note: Users get at most the minimum of (**MAXTRYS**, **LOGFAILURES**) unsuccessful attempts.

IDLEWEEKS=-1 If nonnegative, specify a grace period during which users with expired passwords are allowed to enter a new password. In other words, accounts with expired passwords can stay idle up to this long before being "locked out." If **IDLEWEEKS** is 0, there is no grace period, and expired passwords are the same as invalidated passwords.

PATH= Path for normal users (from /usr/include/paths.h).

SUPATH= Path for superuser (from /usr/include/paths.h).

SYSLOG=FAIL Log to syslog all login failures (**SYSLOG=FAIL**) or all successes and failures (**SYSLOG=ALL**). Log entries are written to the LOG_AUTH facility (see **syslog**(3C) and **syslogd**(1M) for details). No messages are sent to syslog if not set. Note that this is separate from the login log, **/var/adm/loginlog**.

INITGROUPS=YES
If **YES**, make the user session be a member of all of the user's supplementary groups (see **multgrps**(1) or **initgroups**(3C)).

SVR4_SIGNALS=YES
Use the SVR4 semantics for the SIGXCPU and SIGXFSZ signals. If **SVR4_SIGNALS=YES**, the SVR4 semantics are preserved and all processes ignore SIGXCPU and SIGXFSZ by default. If **SVR4_SIGNALS=NO**, these two signals retain their default action, which is to cause the receiving process to core dump. If users intend to make use of the CPU and filesize resource limits, **SVR4_SIGNALS** should be set to **NO**. Note that using these signals while **SVR4_SIGNALS** is set to **YES** causes behavior that varies depending on the login shell. This setting has no affect on processes that explicitly alter the behavior of these signals using the **signal**(2) system call.

SITECHECK= Use an external program to authenticate users instead of using the encrypted password field. This allows sites to implement other means of authentication, such as card keys, biometrics, etc. The program is invoked with user name as the first argument, and remote hostname and username, if applicable. The action taken depend on exit status, as follows:

0 Success; user was authenticated, log in.

1 Failure; exit login.

2 Failure; try again (don't exit login).

other Use normal UNIX authentication.

If authentication fails, the program can chose to indicate either exit code 1 or 2, as appropriate. If the program is not owned by root, is writable by others, or cannot be executed, normal password authentication is performed. It is recommended that the program be given a mode of 500.

Warning: Because this option has the potential to defeat normal IRIX security, any program used in this way must be designed and tested very carefully.

LOCKOUT= If nonzero, after this number of consecutive unsuccessful login attempts by the same user, by all instances of xdm and login, lock the account by invoking **passwd -l username**. Note that this feature allows a denial of service attack that may require booting from the miniroot to fix, as even the root accounts can be locked out.

At some installations, you may be required to enter a dialup password for dialup connections as well as a login password. In this case, the prompt for the dialup password is:

```
Dialup Password:
```

Both passwords are required for a successful login.

For remote logins over the network, **login** prints the contents of **/etc/issue** before prompting for a username or password. The file **/etc/nologin** disables remote logins if it exists; **login** prints the contents of this file before disconnecting the session.

The system can be configured to automate the login process after a system restart. When the file **/etc/autologin** exists and contains a valid user name, the system logs in as the specified user without prompting for a user name or password. The automatic login takes place only after a system restart; once the user logs out, the normal interactive login session is used until the next restart. This is intended to be used at sites where the normal security mechanisms provided by **login** are not needed or desired. If you make five incorrect login attempts, all five are logged in **/var/adm/loginlog** (if it exists) and the TTY line is dropped.

If you do not complete the login successfully within a certain period of time (by default, 20 seconds), you are likely to be silently disconnected.

After a successful login, accounting files are updated, the **/etc/profile** script is executed, the time you last logged in is printed (unless a file **.hushlogin** is present in the user's home directory), **/etc/motd** is printed, the user ID, group ID, supplementary group list, working directory, and command interpreter (usually **sh**) are initialized, and the file **.profile** in the working directory is executed, if it exists. The name of the command interpreter is **–** followed by the last component of the interpreter's pathname (for example, **–sh**). If this field in the password file is empty, the default command interpreter, **/usr/bin/sh** is used. If this field is *****, the named directory becomes the root directory, the starting point for path searches for pathnames beginning with a **/**. At that point **login** is re-executed at the new level which must have its own root structure. At the very least, this root structure must include **/dev/zero**, **/etc/group**, **/etc/passwd**, **/lib/rld**, **/lib/libc.so.1**, **/usr/bin/login**, **/usr/lib/libcrypt.so**, and **/usr/lib/libgen.so**. These files allow **login** to execute correctly, but you also need to include additional files, like shells or applications, that the user is allowed to execute. Since these applications can in turn rely on additional shared libraries, it may also be necessary to place additional shared objects in **/usr/lib**. See the **ftpd**(1M) reference page for more information about setting up a root environment.

The basic *environment* is initialized to:

HOME=*your-login-directory*
LOGNAME=*your-login-name*
PATH=/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/var/mail/*your-login-name*
TZ=*timezone-specification*

The environment can be expanded or modified by supplying additional arguments when **login** prints the prompt requesting the user's login name. The arguments can take either of two forms: *xxx* or *xxx***=***yyy*. Arguments without an equal sign are placed in the environment as

L*n***=***xxx*

where *n* is a number that starts at 0 and is incremented each time a new variable name is required. Variables containing **=** are placed in the environment without modification. If such a variable is already defined, the new value replaces the old value. To prevent users who log in to restricted shell environments from spawning secondary shells that are not restricted, the following environment variables cannot be changed:

HOME
IFS
LOGNAME
PATH

SHELL

Attempts to set environment variables beginning with the following strings (see the **rld**(1) reference page) are ignored, and such attempts are logged via **syslogd**:

_RLD
LD_LIBRARY

login understands simple, single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such characters as spaces and tabs.

To enable dial-in line password protection, two files are required. The file **/etc/dialups** must contain of the name of any dialup ports (for example, **/dev/ttyd2**) that require password protection. These are specified one per line. The second file, **/etc/d_passwd** consists of lines with the following format:

shell:*password*:

This file is scanned when the user logs in, and if the *shell* portion of any line matches the command interpreter that the user gets, the user is prompted for an additional dialin password, which is encoded and compared to that specified in the *password* portion of the line. If the command interpreter cannot be found, the entry for the default shell, **/sbin/sh**, (or, for compatibility with existing configurations, **/bin/sh**) is used. (If both are present, the last one in file is used.) If there is no such entry, no dialup password is required. In other words, the **/etc/d_passwd** entry for **/sbin/sh** is the default.

SHARE II ACTIONS

If the Share II system is installed and enabled, **login** prints the message:

Share login on *ttyname*.

The following privilege and resource checks are made after you have successfully entered your password, but before the initial shell is started:

1. If your **nologin** flag is set, or you are already logged on and your **onelogin** flag is set, you are denied login.

2. If a disk usage exceeds its soft disk limit in any of your domains, a message is printed and you are given a *warning*. If you accumulate too many warnings, further login attempts are denied and you must see your subadministrator to rectify the situation. Whenever you log in or connect by remote shell with no disk usages in excess of any soft limits, all your accumulated warnings are cleared.

3. If you do not have permission to use the terminal, as determined by the respective terminal permission flag, you are denied login.

4. Some installations place limits on terminal connect time, both through logins and remote shell connections. If you have already reached your connect time limit, you are denied login. Otherwise, if the terminal costs more or less to use than normal terminals, its cost is printed. Your remaining connect time is also printed.

If all these checks are passed, **login** proceeds normally.

NOTES

Autologin is controlled by the existence of the **/etc/autologin.on** file. The file is normally created at boot time to automate the login process and then removed by **login** to disable the autologin process for succeeding terminal sessions.

In the default configuration, encrypted passwords for users are kept in the system password file, **/etc/passwd**, which is a text file and is readable by any system user. The program **pwconv**(1M) can be used by the system administrator to activate the shadow password mechanism. When shadow passwords are enabled, the encrypted passwords are kept only in **/etc/shadow**, a file that is only readable by the superuser. Refer to the **pwconv**(1M) reference page for more information about shadow passwords.

FILES

/etc/dialups
/etc/d_passwd
/etc/motd message of the day
/etc/passwd password file
/etc/shadow shadow password file
/etc/profile system profile
$HOME/.profile user's login profile
/usr/lib/iaf/login/scheme
login authentication scheme
/var/adm/lastlog time of last login
/var/adm/loginlog
record of failed login attempts
/var/adm/utmp accounting
/var/adm/wtmp accounting
/etc/default/login
to determine default behavior
/var/mail/*login_name*
mailbox for user *login_name*
/usr/lib/locale/*locale***/LC_MESSAGES/uxcore**
language-specific message file (see **LANG** in **environ**(5))
/etc/limconf the compiled Share II configuration file (machine readable)

SEE ALSO

mail(1), newgrp(1), pwconv(1M), rexecd(1M), rshd(1M), sh(1), su(1M), loginlog(4), passwd(4), profile(4), shadow(4), environ(5), share(5).

DIAGNOSTICS

The message

UX:login: ERROR: Login incorrect

is printed if the user name or the password cannot be matched or if the user's login account has expired or remained inactive for a period greater than the system threshold.

The Share II-specific diagnostic messages are:

Warning *X* **of** *Y*: **soft disk limit exceeded.**
One of your domains has a disk usage in excess of its soft limit.

Connection denied. Too many warnings.
You have reached your warning limit. See your system administrator.

Connection denied. Already logged in – only one login allowed.
You are already logged in at another terminal or connected to the system by remote shell and your **onelogin** flag is set.

Connection denied. Currently barred from logging in.
Your **nologin** flag is set.

Connection denied. No permission to use this terminal.
You are not allowed to log in at this terminal because of a clear **terminal permission** flag.

Share login on *ttyname* – **terminal cost is** *X* **times normal.**
You are charged for use of this terminal at *X* times the rate of a normal terminal.

You have a remaining terminal connect time of *Y*.
You may use this terminal until you have used up your remaining connect time, at which point you are forced to log out.

Connection denied. Terminal connect time limit exceeded.
You have already reached your terminal connect time limit.

Share not configured – no limit checks.
The configuration file is unreadable for some reason, so terminal privileges, connect time limits, and disk space limits could not be checked.

NAME

lvck – check and restore consistency of logical volumes

SYNOPSIS

/sbin/lvck [**–l** lvtabname] [**lv**x]
/sbin/lvck **–d**
/sbin/lvck block_special_filename

DESCRIPTION

lvck checks the consistency of *logical volumes* by examining the logical volume labels of devices constituting the volumes. Depending on the invocation, the volumes can also be checked against *lvtab* entries, see *lvtab*(4). The default system file */etc/lvtab* is normally assumed in this case; an alternate *lvtab* file can be specified with the **–l** option.

Invoked without parameters, *lvck* checks every logical volume for which there is an entry in */etc/lvtab*.

Invoked with the name of a logical volume device, for example **lv0**, *lvck* checks only that entry in */etc/lvtab*.

Invoked with the **–d** flag, *lvck* ignores */etc/lvtab* and searches through all disks connected to the system to locate all logical volumes that are present. *lvck* prints a description of each logical volume found in a form resembling an *lvtab* entry; this facilitates recreation of an *lvtab* for the system should this be necessary.

Invoked with the device block special filename of a disk device (for example, */dev/dsk/ips0d1s4*), *lvck* prints any logical volume label that exists for that device, again in a form resembling an *lvtab* entry. This mode of *lvck* is purely informational; no checks are made of any other devices mentioned in the label.

lvck has some repair capabilities. If it determines that the only inconsistency in a logical volume is that a minority of devices have missing or corrupt labels, it is able to restore a consistent logical volume by rewriting good labels. *lvck* interactively queries the user before attempting any repairs on a volume.

DIAGNOSTICS

lvck detects four general types of errors:

1) Disks connected in the wrong place.

2) Inconsistencies between the on-disk labels of a volume.

3) Internal inconsistencies in an *lvtab* entry.

4) Inconsistencies between a volume defined by its on-disk labels and the *lvtab* entry for that volume.

(The two latter are relevant only for modes of *lvck* that examine the *lvtab*.) Details of these errors are given below.

1) If a disk device that is a member of a logical volume is connected with the wrong id, the volume cannot be used since the device is not correctly located from the labels. *lvck* prints a message describing the problem. For example:

```
lvck: device currently connected as /dev/dsk/ips1d2s7
was initialized when connected as /dev/dsk/ips0s2s7.

lvck: Incorrect device connections must be rectified
before logical volumes can be used.
```

The offending disks must be physically reconnected with the appropriate ID. See *intro*(7) for details of the SGI disk device ID conventions.

2) If *lvck* detects inconsistencies between the on-disk labels for a logical volume, it prints a description of the volume in a form resembling an *lvtab* entry, with the device pathname for each member on a separate line. A short message describing the problem with the member appears on the line with the pathname. For example:

```
lv6:test 6:stripes=3:step=31:devs= \
    /dev/dsk/ips0d1s2,     \
    /dev/dsk/ips0d1s3,  <CAN'T ACCESS>   \
    /dev/dsk/ips0d1s4,     \
    /dev/dsk/ips0d1s11, <NO LABEL PRESENT>   \
    /dev/dsk/ips0d1s12,    \
    /dev/dsk/ips0d1s13
```

Possible messages and their meanings are:

<NOT A MEMBER OF THIS VOLUME>
: The label for this device identifies it as a member of a different volume from the other devices. Probably the wrong disk has been connected.

<CAN'T ACCESS>
: The named special file is missing or cannot be opened. The disk may be missing, or there may be a hardware problem.

<CAN'T READ DISK HEADER>
: *lvck* could not read the disk header partition for this device to search for the logical volume label.

<ILLEGAL PARTITION TYPE>
: The pathname refers to a type of partition (such as track replacement) that is not legal as part of a logical volume. Probably logical volume labels or disk headers have been corrupted.

<INCORRECT PARTITION SIZE>
The partition size does not agree with the logical volume label. Possibly the disk has been incorrectly repartitioned since creation of the logical volume.

<SPECIAL FILE DEV IS WRONG>
The major and minor numbers of the special file disagree with the SGI naming conventions. There has probably been an incorrect use of *mknod*(1M) in the */dev/dsk* directory.

<FILE NOT BLOCK SPECIAL>
The pathname does not refer to a block special file, even though it has the conventional format. The /dev/dsk directory needs repair.

<INCORRECT PARTITION TYPE>
The partition type stored in the disk header does not indicate that this partition is a logical volume member. Probably the wrong disk has been connected.

<LABEL UNREADABLE>
The disk header directory indicates that a logical volume label exists for this device, but it cannot be read. Possibly there is a bad block on the disk.

<NO LABEL PRESENT>
There is no logical volume label for this device. It may have been inadvertently deleted, or the wrong disk may have been connected.

<LABEL CORRUPTED>
Label is present but damaged.

3) Internal inconsistencies in *lvtab* entries. In this case *lvck* prints error messages that are intended to be self-explanatory. Possible messages are listed below (the items in brackets <xxx> represent places where the actual erring values appear):

```
Lvtab entry with no device name: ignored

Lvtab entry with illegal device name <xxx>: ignored

Illegal number of pathnames <n> in lvtab entry <lvx>: ignored

Number of pathnames in lvtab entry <lvx> is not multiple of
striping: entry ignored.

Illegal striping step in lvtab entry <lvx>: ignored

Duplicate lvtab entry <lvx>: ignored

Bad pathname <xxx> in lvtab entry <lvx>: entry ignored.
```

```
Duplicate pathname <xxx> in lvtab entry <lvx>: entry ignored

Illegal entry <lvx> in logical volume table: ignored
```

See *lvtab*(4) for details of the expected form of *lvtab* entries, and constraints upon the entries. Limits on the number of devices in a volume, striping step size, volume name length, and so forth, are given in the include file <*sys/lvtab.h*>.

4) Inconsistencies between on-disk labels and the *lvtab* entry. This can occur if the *lvtab* entry has been incorrectly modified or if the disks connected to the system do not contain the logical volume expected in the *lvtab* entry. *lvck* prints error messages that are intended to be self-explanatory. Possible messages are listed below (the items in brackets <xxx> represent places where the actual erring values appear):

```
Volume name <xxx> in lvtab entry disagrees with name <yyy>
in on-disk labels.

Number of devs in lvtab entry <lvx> is greater than number <d>
in on-disk labels.

Number of devs in lvtab entry <lvx> is less than number <d> in
on-disk labels.

Stripes specified in lvtab entry for <lvx> don't agree with
on-disk labels.

Step specified in lvtab entry for <lvx> doesn't agree with
on-disk labels.
```

SEE ALSO

lvinit(1M), mklv(1M), lvtab(4), intro(7), lv(7M).

CAVEAT

The repair capabilities of *lvck* are limited to recreating damaged or missing logical volume labels for disk devices so that the system is again able to use an ensemble of disk devices as a logical volume. However, if data on the devices themselves has been corrupted or if an incorrect disk device has been connected, the contents of the logical volume is corrupt. It is **strongly** advisable to check that no incorrect disks have been connected before proceeding with any repair attempt.

NAME

lvinfo – print information about active logical volumes

SYNOPSIS

lvinfo [pathname | volume_device_name]

DESCRIPTION

lvinfo prints descriptions of logical volumes that are currently active in a system. See *lv*(7M).

Invoked without arguments, *lvinfo* prints descriptions of all volumes that are currently active. These may be only a subset of those described in the *lvtab*(4) configuration file, because volumes may have failed to initialize. Failure to initialize could be due to missing physical disks, for example.

The information printed can be limited to specified logical volume devices by using options. These can be either the full pathname of a logical volume device, such as */dev/rdsk/lv1*, or a logical volume device name of the form *lvn* where *n* is a small integer. For example, *lv2*.

The information printed by *lvinfo* consists of a line giving the total size of the volume in 512-byte basic blocks, followed by a description of the volume in a form, which resembles an entry in *lvtab*(4).

EXAMPLE

```
prompt> /sbin/lvinfo
# lv5 size is 4680648 blocks
lv5: :stripes=3:step=148:devs= /dev/dsk/ipi0d1s6, \
        /dev/dsk/ipi0d8s6, \
        /dev/dsk/ipi1d0s6
```

Note that the values of all options are printed, even though some may have been omitted in the *lvtab* entry, causing defaults to be used.

Also note that the human-readable name of the volume is not printed: *lvinfo* works from information held by the *lv*(7M) device driver, which does not use this information.

DIAGNOSTICS

Invoked with no arguments, *lvinfo* simply prints the active volumes and is silent about any volumes that are not initialized.

To obtain a printout of all logical volume devices whether initialized or not, invoke as:

```
lvinfo /dev/rdsk/lv*
```

Invoked with arguments, it prints for each argument either a logical volume description as above or an error message. Possible error messages are:

XXXX `is not a logical volume device name.`
lvxx `is not initialized.`
These messages means that there is a special device file for a logical volume of that name, but it is not currently active. This might be because there is no entry in */etc/lvtab* for that volume or because initialization of that volume failed, owing to faulty or missing disks, for example.

`Can't open /dev/rdsk/lv`*x*
This message means that the given logical volume name is legal and plausible, but no special device file exists for it. Probably a volume of that name has never been created on the system.

SEE ALSO

lvck(1M), lvinit(1M), mklv(1M), lvtab(4), lv(7M).

NAME

lvinit – initialize logical volume devices

SYNOPSIS

/sbin/lvinit [**–l** lvtabname] [volume_device_name ...]

DESCRIPTION

lvinit initializes the logical volume device driver which allows access to disk storage as *logical volumes* (see *lv*(7M)). It is run automatically on system startup, and will not normally need to be invoked explicitly. It works from entries in */etc/lvtab* (see *lvtab*(4)).

No data access to a logical volume device is possible until it has been initialized with *lvinit*, because the initialization process provides the driver with the information needed to map requests on the logical volume device into requests on the underlying physical disk devices. This implies that the root filesystem of a machine must reside on a regular partition rather than a logical volume, because *lvinit* must be accessible before logical volumes can be initialized.

Invoked without arguments, *lvinit* initializes every logical volume device for which there is an entry in */etc/lvtab*. The **–l** option allows an alternate *lvtab* file to be specified. If *volume_device_name* arguments are present, it initializes only those volumes in the *lvtab* identified by the arguments. These arguments must be of the form **lv***n*, where *n* is a small integer. For example, **lv2**.

The necessary information about the devices (disk partitions) constituting the logical volume, and the volume geometry, is obtained from the logical volume labels for the devices specified in the *lvtab* entry as constituting the volume.

The constituent devices must have been labeled as members of the volume with *mklv* before the volume can be initialized.

DIAGNOSTICS

lvinit does checks of the *lvtab* entry and the on-disk labels of a volume before attempting initialization.

See the DIAGNOSTICS section of the *lvck* reference page for possible error messages.

SEE ALSO

lvck(1M), mklv(1M), lvtab(4), lv(7M).

NAME

MAKEDEV – create device special files

SYNOPSIS

/dev/MAKEDEV [*target*] [*parameter=val*]

DESCRIPTION

MAKEDEV creates specified device files in the current directory; it is primarily used for constructing the */dev* directory. It is a "makefile" processed by the *make*(1) command. Its arguments can be either targets in the file or assignments overriding parameters defined in the file. The *target*s **alldevs** and **owners** are assumed if no other targets are present (see below).

All devices are created relative to the current directory, so this command is normally executed from */dev*. In order to create the devices successfully, you must be the superuser.

The following are some of the *target* arguments that are recognized by *MAKEDEV*. For a complete list you may need to examine the script.

ttys Creates *tty* (controlling terminal interface) files for CPU serial ports. In addition, creates special files for *console*, *syscon*, *systty*, *keybd*, *mouse*, *dials*, and *tablet*. See *duart*(7), *console*(7), *keyboard*(7), *mouse*(7), *pckeyboard*(7), and *pcmouse*(7) for details.

cdsio Creates additional *tty* files enabled by using the Central Data serial board.

pty Creates special files to support "pseudo terminals." This target makes a small number of files, with more created as needed by programs using them. Additional *pty* files can be made for older programs not using library functions to allocate *pty*s by using the *parameter* override *MAXPTY=100*, or any other number between 1 and 199. See *pty*(7M) for details.

dks Creates special files for SCSI disks. See *dks*(7M) for details.

rad Creates special files for SCSI attached RAID disks. See *raid*(1M) and *usraid*(7M) for details.

fds Creates special files for SCSI floppy drives. See *smfd*(7M) for details.

usrvme Creates special files for user level VME bus adapter interfaces. See *usrvme*(7M) for details.

usrdma Creates special files for user level access to DMA engines. See *usrdma*(7M) for details.

tps Creates special files for SCSI tape drives. See *tps*(7M) for details.

hl Creates special files for the hardware spinlock driver to use in process synchronization (IRIS-4D/GTX models only).

t3270 Creates the special files for the IBM 3270 interface controller.

gse Creates the special files for the IBM 5080 interface controller.

dn_ll Creates the special file for the 4DDN logical link driver.

dn_netman Creates the special file for the 4DDN network management driver.

audio Creates the special file for the bi-directional audio channel interface for the IRIS-4D/20 series. See *audio*(1) for details.

plp Creates the special file for the parallel printer interface for the IRIS-4D/20 series. See *plp*(7) for details.

ei Creates the special file for the Challenge/Onyx external interrupt interface. See *ei*(7) for details.

generic Creates miscellaneous, commonly used devices: *tty*, the controlling terminal device; *mem*, *kmem*, *mmem*, and *null*, the memory devices; *prf*, the kernel profiling interface; *tport*, the texport interface; *shmiq*, the event queue interface; *gfx*, *graphics*, the graphics device interfaces; and *zero*, a source of zeroed unnamed memory. See *tty*(7), *mem*(7), *prf*(7), and *zero*(7) for details concerning some of these respective devices.

links This option does both *disk* and *tape*

disk This option creates all the disk device special files for the *dks* drives, and then creates links by which you can conveniently reference them without knowing the configuration of the particular machine. The links *root*, *rroot*, *swap*, *rswap*, *usr*, *rusr*, *vh*, and *rvh* are created to reference the current root, swap, usr and volume header partitions.

tape This option creates all the *tps* tape devices, then makes links to *tape*, *nrtape*, *tapens*, and *nrtapens* for the first tape drive found, if one exists. It checks for SCSI in descending target ID order, and ascending SCSI bus number.

mindevs This option is shorthand for creating the *generic*, *links*, *pty*, *ttys*, device files.

alldevs This option creates all of the device special files listed above.

owners This option changes the owner and group of the files in the current directory to the desired default state.

onlylinks This option does only the link portion of *disk* and *tape* above, in case a different disk is used as root, or a different tape drive is used.

BUGS

The links made for */dev/usr* and */dev/rusr* always point to partition 6 of the root drive. While this is the most common convention, it is not invariable.

If a system has been reconfigured with the */usr* filesystem in some place other than this default, by specifying the device in */etc/fstab* (see *fstab*(4)), the */dev/usr* and */dev/rusr* devices will NOT point to the device holding the real */usr* filesystem.

SEE ALSO

install(1), make(1), mknod(1M).

NAME

mkfs – construct a filesystem

SYNOPSIS

mkfs [**–t efs**] efs_mkfs_options
mkfs [**–t xfs**] xfs_mkfs_options

DESCRIPTION

mkfs constructs a filesystem by writing on the special file given as one of the command line arguments. The filesystem constructed is either an EFS filesystem or an XFS filesystem depending on the arguments given. *mkfs* constructs EFS filesystems by executing *mkfs_efs*(1M); XFS filesystems are constructed by executing *mkfs_xfs*(1M).

The filesystem type chosen can be forced with the **–t** option (also spelled **–F**). If one of those options is not given, *mkfs* determines which filesystem type to construct by examining its arguments. If *mkfs special* is given, then the choice is EFS unless the special device is an XLV volume with a log subvolume.

SEE ALSO

mkfs_efs(1M), mkfs_xfs(1M).

NAME

mkfs_efs – construct an EFS filesystem

SYNOPSIS

mkfs_efs [**–q**] [**–a**] [**–i**] [**–r**] [**–n** inodes] special [proto]
mkfs_efs [**–q**] [**–i**] [**–r**] special blocks inodes heads sectors cgsize cgalign ialign [proto]

DESCRIPTION

mkfs_efs constructs a filesystem by writing on the *special* file using the values found in the remaining arguments of the command line. Normally *mkfs_efs* prints the parameters of the filesystem to be constructed; the **–q** flag suppresses this.

If the **–i** flag is given, *mkfs_efs* asks for confirmation after displaying the parameters of the filesystem to be constructed.

In it's simplest (and most commonly used form), the size of the filesystem is determined from the disk driver. As an example, to make a filesystem on partition 7 (all of the useable portion of an option drive, normally) on drive 7 on SCSI bus 0, use:

```
mkfs_efs /dev/rdsk/dks0d7s7
```

The **–r** flag causes *mkfs_efs* to write only the superblock, without touching other areas of the filesystem. See the section below on the recovery option.

The **–a** flag causes *mkfs_efs* to align inodes and data on cylinder boundaries (equivalent to setting cgalign and ialign to a cylinder size). This option can result in a loss of 10MB or more in a filesystem, since the resulting cylinder groups are not very flexible in size, and runt cylinder groups are not allowed. Aligning data and inodes with this option can result in an increase in performance (about two percent) on drives that have a fixed number of sectors per track. Many SCSI disk drives do not have a fixed number of sectors per track, and thus, will not see a benefit from this option.

When the first form of *mkfs_efs* is used, *mkfs_efs* obtains information about the device size and geometry by means of appropriate IOCTLs, and assigns values to the filesystem parameters on the basis of this information.

If the **–n** option is present, however, the given number of inodes is used rather than the default. This allows a nonstandard number of inodes to be assigned without needing to resort to the long form invocation.

If the second form of *mkfs_efs* is used, then all the filesystem parameters must be specified from the command line. Each argument other than *special* and *proto* is interpreted as a decimal number.

The filesystem parameters are:

blocks The number of *physical* (512-byte) disk blocks the filesystem will occupy. The current maximum limit on the size of an EFS filesystem is 16777214 blocks (two to the 24th power). This can also be expressed as 8 gigabytes. *mkfs_efs* does not attempt to make a filesystem larger than this limit.

inodes The number of inodes the filesystem should have as a minimum.

heads An unused parameter, retained only for backward compatibility.

sectors The number of sectors per track of the physical medium.

cgsize The size of each cylinder group, in disk blocks, approximately.

cgalign The boundary, in disk blocks, that a cylinder group should be aligned to.

ialign The boundary, in disk blocks, that each cylinder group's inode list should be aligned to.

Once *mkfs_efs* has the filesystem parameters it needs, it then builds a filesystem containing two directories. The filesystem's root directory is created with one entry, the *lost+found* directory. The *lost+found* directory is filled with zeros out to approximately 10 disk blocks, so as to allow space for *fsck*(1M) to reconnect disconnected files. The boot program block, block zero, is left uninitialized.

If the optional *proto* argument is given, *mkfs_efs* uses it as a prototype file and takes its directions from that file. The blocks and inodes specifiers in the *proto* file are provided for backwards compatibility, but are otherwise unused. 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   fifo p--644 3 1
11   slink     l--644 3 1 /a/symbolic/link
12   :  This is a comment line
13   $
14.  $
```

Line 1 is a dummy string. (It was formerly the bootfilename.) It is present for backward compatibility;

boot blocks are not used on SGI systems, and *mkfs_efs* merely clears block zero.

Note that some string of characters must be present as the first line of the proto file to cause it to be parsed correctly; the value of this string is immaterial since it is ignored.

Line 2 contains two numeric values (formerly the numbers of blocks and inodes). These are also merely for backward compatibility: two numeric values must appear at this point for the proto file to be correctly parsed, but their values are immaterial since they are ignored.

Lines 3-11 tell *mkfs_efs* about files and directories to be included in this filesystem.

Line 3 specifies the root directory.

lines 4-6 and 8-10 specifies other directories and files. Note the special symbolic link syntax on line 11.

The **$** on line 7 tells *mkfs_efs* to end the branch of the filesystem it is on, and continue from the next higher directory. It must be the last character on a line. The **:** on line 12 introduces a comment; all characters up until the following newline are ignored. Note that this means you cannot have files in a prototype file whose name contains a **:**. The **$** on lines 13 and 14 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 six-character string. The first character specifies the type of the file. The character range is **−bcdpl** to specify regular, block special, character special, directory files, named pipes (fifos) and symbolic links, 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 six 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 can be a pathname whence the contents and size are copied. If the file is a block or character special file, two decimal numbers follow that give the major and minor device numbers. If the file is a symbolic link, the next token of the specification is used as the contents of the link. If the file is a directory, *mkfs_efs* 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 **$**.

RECOVERY OPTION

The **−r** flag causes *mkfs_efs* to write only the superblock, without touching the remainder of the filesystem space. This allows a last-ditch recovery attempt on a filesystem whose superblocks have been destroyed: by running *mkfs_efs* on the device with the **−r** option, a superblock is created from which *fsck*(1M) can obtain the geometry information it needs to analyze the filesystem.

Note that this procedure is only of use if the regenerated superblock matches the parameters of the original filesystem. If the filesystem was created using the long form invocation, parameters identical to the original invocation must be given with the **–r** option. Note also that filesystem defaults may change from release to release to allow more efficient use of newer disk technologies; thus, the **–r** option may not be useful for filesystems created under IRIX versions other than the version being run.

It should be clear that this is a limited recovery facility; it does not help if, for example, the root directory of the filesystem has been destroyed.

SEE ALSO

chmod(1), mkfp(1M), mkfs(1M), mkfs_xfs(1M), dir(4), efs(4).

BUGS

With a prototype file, it is not possible to specify hard links.

NAME

mkfs_xfs – construct an XFS filesystem

SYNOPSIS

mkfs_xfs [**–b** subopt=value] [**–d** subopt[=value]] [**–i** subopt=value] [**–l** subopt[=value]] [**–p** protofile] [**–q**] [**–r** subopt[=value]] device

DESCRIPTION

mkfs_xfs constructs an XFS filesystem by writing on a special file using the values found in the arguments of the command line. It is invoked automatically by *mkfs*(1M) when *mkfs* is given the **–t xfs** option or options that are specific to XFS.

In it's simplest (and most commonly used form), the size of the filesystem is determined from the disk driver. As an example, to make a filesystem on partition 7 (all of the useable portion of an option drive, normally) on drive 7 on SCSI bus 0, with an internal log, use:

```
mkfs_xfs /dev/rdsk/dks0d7s7
```

XFS filesystems are composed of a data section, a log section, and optionally a real-time section. This separation can be accomplished using the XLV volume manager to create a multi-subvolume volume, or by embedding an *internal* log section in the data section. In the former case, the *device* name is supplied as the final argument. In the latter case a disk partition, *lv*(7M) logical volume, or XLV logical volume without a log subvolume can contain the XFS filesystem, which is named by the **–d name=***special* option or by the final argument.

Each of the *subopt=value* elements in the argument list above can be given as multiple comma-separated *subopt=value* suboptions if multiple suboptions apply to the same option. Equivalently, each main option can be given multiple times with different suboptions. For example, **–l internal,size=1000b** and **–l internal –l size=1000b** are equivalent.

In the descriptions below, sizes are given in bytes, blocks, kilobytes, or megabytes. Sizes are treated as hexadecimal if prefixed by 0x or 0X, octal if prefixed by 0, or decimal otherwise. If suffixed with **b** then the size is converted by multiplying it by the filesystem's block size. If suffixed with **k** then the size is converted by multiplying it by 1024. If suffixed with **m** then the size is converted by multiplying it by 1048576 (1024 * 1024).

–b Block size options.

This option specifies the fundamental block size of the filesystem. The valid suboptions are: **log=***value* and **size=***value*; only one can be supplied. The block size is specified either as a base two logarithm value with **log=**, or in bytes with **size=**. The default value is 4096 bytes (4 KB). The minimum value for block size is 512; the maximum is 65536 (64 KB).

–d Data section options.

These options specify the location, size, and other parameters of the data section of the filesystem. The valid suboptions are: **agcount=***value*, **file**[**=***value*], **name=***value*, and **size=***value*.

The **agcount** suboption is used to specify the number of allocation groups. The data section of the filesystem is divided into allocation groups to improve the performance of XFS. More allocation groups imply that more parallelism can be achieved when allocating blocks and inodes. The minimum allocation group size is 16 MB; the maximum size is just under 4 GB. The data section of the filesystem is divided into *agcount* allocation groups (default value 8, unless the filesystem is smaller than 128 MB or larger than 32 GB).

The **name** suboption can be used to specify the name of the special file containing the filesystem. In this case, the log section must be specified as **internal** (with a size, see the **–l** option below) and there can be no real-time section. Either the block or character special device can be supplied. An XLV logical volume with a log subvolume cannot be supplied here. Note that the default log in this case is a 1000 block internal log.

The **file** suboption is used to specify that the file given by the **name** suboption is a regular file. The suboption value is either 0 or 1, with 1 signifying that the file is regular. This suboption is used only to make a filesystem image (for instance, a miniroot image). If the value is omitted then 1 is assumed.

The **size** suboption is used to specify the size of the data section. This suboption is required if **–d file[=1]** is given. Otherwise, it is only needed if the filesystem should occupy less space than the size of the special file.

–i Inode options.

This option specifies the inode size of the filesystem, and other inode allocation parameters. The XFS inode contains a fixed-size part and a variable-size part. The variable-size part, whose size is affected by this option, can contain: directory data, for small directories; symbolic link data, for small symbolic links; the extent list for the file, for files with a small number of extents; and the root of a tree describing the location of extents for the file, for files with a large number of extents.

The valid suboptions for specifying inode size are: **log=***value*, **perblock=***value*, and **size=***value*; only one can be supplied. The inode size is specified either as a base two logarithm value with **log=**, in bytes with **size=**, or as the number fitting in a filesystem block with **perblock=**. The default value is 256 bytes. The minimum value for inode size is 128, and the maximum value is 2048 (2 KB) subject to the restriction that the inode size cannot exceed one half of the filesystem block size.

The option **maxpct=***value* specifies the maximum percentage of space in the filesystem that can be allocated to inodes. The default value is 25%. Setting the value to 0 means that essentially all of the filesystem can become inode blocks.

–l Log section options.

These options specify the location, size, and other parameters of the log section of the filesystem. The valid suboptions are: **internal[=***value***]** and **size=***value*.

The **internal** suboption is used to specify that the log section is a piece of the data section instead of being a separate part of an XLV logical volume. The suboption value is either 0 or 1, with 1 signifying that the log is internal. If the value omitted, 1 is assumed.

The **size** suboption is used to specify the size of the log section. This suboption is required if **–l internal[=1]** is given. Otherwise, it is only needed if the log section of the filesystem should occupy less space than the size of the special file.

For a filesystem which is not contained in an XLV logical volume with a log subvolume, the default is to make an internal log 1000 blocks long.

–p *protofile*

If the optional **–p** *protofile* argument is given, *mkfs_xfs* uses *protofile* as a prototype file and takes its directions from that file. The blocks and inodes specifiers in the *protofile* are provided for backwards compatibility, but are otherwise unused. The prototype file contains tokens separated by spaces or newlines. 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      fifo    p--644 3 1
11      slink   l--644 3 1 /a/symbolic/link
12      :  This is a comment line
13      $
14      $
```

Line 1 is a dummy string. (It was formerly the bootfilename.) It is present for backward compatibility; boot blocks are not used on SGI systems.

Note that some string of characters must be present as the first line of the proto file to cause it to be parsed correctly; the value of this string is immaterial since it is ignored.

Line 2 contains two numeric values (formerly the numbers of blocks and inodes). These are also merely for backward compatibility: two numeric values must appear at this point for the proto file to be correctly parsed, but their values are immaterial since they are ignored.

Lines 3-11 tell *mkfs_xfs* about files and directories to be included in this filesystem. Line 3 specifies the root directory. Lines 4-6 and 8-10 specifies other directories and files. Note the special symbolic link syntax on line 11.

The **$** on line 7 tells *mkfs_xfs* to end the branch of the filesystem it is on, and continue from the next higher directory. It must be the last character on a line. The colon on line 12 introduces a comment; all characters up until the following newline are ignored. Note that this means you cannot have a file in a prototype file whose name contains a colon. The **$** on lines 13 and 14 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 **−bcdpl** to specify regular, block special, character special, directory files, named pipes (fifos), and symbolic links, respectively. The second character of the mode is either **u** or − to specify setuserID mode or not. The third is **g** or − for the setgroupID 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 IDs of the owner of the file.

If the file is a regular file, the next token of the specification can be a pathname from which the contents and size are copied. If the file is a block or character special file, two decimal numbers follow that give the major and minor device numbers. If the file is a symbolic link, the next token of the specification is used as the contents of the link. If the file is a directory, *mkfs_xfs* 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 **$**.

−q Quiet option.

Normally *mkfs_xfs* prints the parameters of the filesystem to be constructed; the **−q** flag suppresses this.

–r Real-time section options.

These options specify the location, size, and other parameters of the real-time section of the filesystem. The valid suboptions are: **extsize=***value* and **size=***value*.

The **extsize** suboption is used to specify the size of the blocks in the real-time section of the filesystem. This size must be a multiple of the filesystem block size. The minimum allowed value is the filesystem block size or 4 KB (whichever is larger); the default value is 64 KB; the maximum allowed value is 1 GB. The real-time extent size should be carefully chosen to match the parameters of the physical media used.

The **size** suboption is used to specify the size of the real-time section. This suboption is only needed if the real-time section of the filesystem should occupy less space than the size of the XLV real-time subvolume.

SEE ALSO

mkfs(1M), mkfs_efs(1M).

BUGS

With a prototype file, it is not possible to specify hard links.

NAME

mklv – construct or extend a logical volume

SYNOPSIS

/sbin/mklv [**–l** lvtabname] [**–f**] volume_device_name

DESCRIPTION

mklv constructs a logical volume by writing logical volume labels for the devices that are to constitute the volume. It works from an entry in */etc/lvtab* (see *lvtab*(4)) and constructs the logical volume identified in */etc/lvtab* by the *volume_device_name* argument. This argument must be of the form **lv***n* where *n* is a small integer, for example **lv2**.

mklv obtains the necessary information about the devices (disk partitions) constituting the logical volume and the volume geometry, from the *lvtab* entry.

mklv also creates the necessary device files for the logical volume in the */dev/dsk* and */dev/rdsk* directories, if these files do not already exist.

An existing logical volume can be extended by adding further device pathnames to the end of the existing */etc/lvtab* entry, and then rerunning *mklv* on that entry.

After writing the labels, *mklv* initializes the logical volume device with the appropriate information. Effectively this invokes the functionality of *lvinit*(1M).

The following options are accepted by *mklv*.

–f Normally, for safety, *mklv* checks whether any of the specified constituent devices are already part of a logical volume, or appear to contain a filesystem. These checks are skipped if the **–f** flag is given; this is useful for recycling disks that contain obsolete logical volumes or filesystems.

Note that this flag does **not** override a check for mounted filesystems on any of the specified devices. *mklv* unconditionally refuses to incorporate any device containing a mounted filesystem as part of a logical volume.

–l *mklv* normally works from the default system file */etc/lvtab*. This option allows an alternate *lvtab* file to be specified.

DIAGNOSTICS

1) If the *volume_device_name* argument is not found in the *lvtab* file, *mklv* prints the error message:

```
mklv: <arg> not found in lvtab.
```

2) The *lvtab* entry is checked before use. *mklv* prints error messages if problems are found. See the DIAGNOSTICS section of *lvck*(1M) for possible error messages concerned with *lvtab* entries.

3) *mklv* checks the specified devices for accessibility and legality before proceeding. If errors are detected *mklv* prints the *lvtab* entry, with each device pathname on a separate line. A short message describing the problem with the device appears on the line with the pathname. For example:

```
lv6:test 6:stripes=3:step=31:devs= \
    /dev/dsk/ips0d1s2,     \
    /dev/dsk/ips0d2s3,  <CAN'T ACCESS>   \
    /dev/dsk/ips0d3s4,  <WRONG PARTITION SIZE>    \
    /dev/dsk/ips1d1s11, <CAN'T READ DISK HEADER>   \
    /dev/dsk/ips0d1s8,  <ILLEGAL PARTITION TYPE>   \
    /dev/dsk/ips0d1s13
```

Possible messages and their meanings are listed in the DIAGNOSTICS section of *lvck*(1M).

4) *mklv* checks for the existence of a mounted filesystem on any of the specified devices. If so, it exits with the error message:

```
<pathname> contains a mounted filesystem.
```

5) If the **–f** flag is not given, *mklv* checks whether there appears to be an unmounted filesystem on any of the specified devices. If so, it prints the warning:

```
<pathname> appears to contain a filesystem.
This will be wiped out if we proceed. OK? (y/n)
```

and waits for user response before proceeding.

As the message implies, any previous filesystem on the disks is erased. This avoids errors resulting from erroneous attempts to mount individual disks that are now part of a logical volume.

There are cases where an existing filesystem should be retained: when a volume is being extended, or when a disk partition containing a filesystem is being made into a volume for extension. These cases are detected by *mklv*. No message or filesystem modification occurs on the relevant disk.

6) If the **–f** flag is not given, *mklv* checks whether any of the specified devices are already part of a logical volume that is inconsistent with the volume specified in the *lvtab* entry. (It is legal when extending a volume for the on-disk volume to be a consistent subset of the newly specified volume.) If an inconsistency exists, *mklv* prints the error message:

```
devices specified for <lvx> already contain a logical volume
which is inconsistent with the volume specified.
```

NOTES

1) Execution of *mklv* does not cause a filesystem to be placed on the volume: it simply creates the volume, which can be regarded as effectively a large disk. If you want to use the volume for filesystem storage, a filesystem must be placed on it (see *mkfs*(1M)).

2) The logical volume labels do not occupy space on the constituent partitions themselves, but are files in the Disk Volume Header partitions of the disks containing the partitions, located via the header directory (see *vh*(7M)). They are named for the partitions to which they refer, having names of the form lvlabn where n is the partition number. Thus, if partition 6 on a disk is part of a logical volume, there is a logical volume label file with the name *lvlab6* in the header partition of that disk.

DEVICE NAME LINKS

Administrators sometimes make links in the */dev* directory to allow disk devices to be referenced by shorter names.

Some caution is needed, however, since the script *MAKEDEV*(1M), which is run on every system installation, removes certain links and replaces them with system defaults. In particular, the link */dev/usr* should never be changed to refer to a logical volume. If the */usr* filesystem is moved to a logical volume, the *fstab*(4) entry for */usr* should be changed to refer explicitly to the appropriate logical volume device.

SEE ALSO

growfs(1M), lvck(1M), lvinit(1M), lvtab(4), lv(7M).

NAME

ml – load dynamic kernel modules

SYNOPSIS

ml list [–**rlb**]
ml [**ld**|**reg**] [**-d**] [**-v**] –[**cbBfmljir**] module.o –**p** prefix
[–**s** major major ...] [–**a** modname] [–**t** autounload_delay]
ml [**unld**|**unreg**] [**-v**] id ...
ml debug [**-vsn**]

DESCRIPTION

The *ml* command provides a means of loading and unloading dynamic kernel modules. The first argument to *ml* specifies its action from one of the following: list, load, unload, register, unregister. With no options, *ml* acts as if it were invoked as **ml list –b**. The **ld**, **unld**, **reg**, and **unreg** options are available only to the superuser. If successful, the *ml* command executes silently, unless the **–v** option is specified.

ml list provides a list of modules that are currently known by the kernel. The following options to **ml list** are recognized:

–r Print registered modules only.

–l Print loaded modules only.

–b Print both loaded and registered modules.

ml ld causes a kernel module to be loaded into memory and prepared to be executed as part of the kernel. **ml reg** is similar to **ml ld**, except that the module is not loaded until the first time the module is opened. If the **–v** option is specified, **ml ld** and **ml reg** list a module ID number on standard output that can be used for subsequent unloading. The **–d** option allows the module's static symbols to be added to its symbol table. This allows symmon to have access to all of the module's symbols and is useful for debugging. The module type must be specified by one of the following options to **ml ld** and **ml reg**:

–c Module is a character device driver.

–b Module is a block and character device driver.

–B Module is a block device driver only.

–f Module is a streams device driver.

–m Module is a pushable streams module.

–l Module is a library module.

−j Module is a filesystem.

−i Module is the kernel debug module.

−r Module is a symbol table module.

The argument following the module type must be the name of the object file for the module.

With the exception of the kernel debug module, a module prefix must be specified with the **-p** option. The module prefix is the string used to find the various entrypoints within a module.

Modules of type **c**, **b**, **B**, or **f** can specify an external device major number with the **−s** option. If a specific major number is not requested, one is provided from the unused major numbers in the system. If the major number is already in use, the module is not loaded.

Modules of type **m** can provide a streams module name with the **-a** option that is entered into the **fmodsw** structure in the kernel. If a streams module name is not provided, the module is given the same name as the object file, with any trailing **.o** removed. If the module name is already in use, the module is not loaded.

Modules of type **i** are special kernel debugging modules to be used in conjunction with the **idbg**(1M) command.

Modules of type **r** are symbol table modules. A symbol table is created from the ELF symbol information in the file specified. This symbol table can be used by other modules to link against when loaded. A kernel runtime symbol table can be created manually using this command. For more information, see *mload*(4).

Modules that are registered are automatically auto-unloaded after last close by using a default auto-unload delay that is systuneable. A specific delay can be specified, in minutes, using the **-t** option. Modules can also be configured to not be auto-unloaded by using **-t −2**.

ml unld unloads the loaded kernel modules specified by *id*. Likewise, **ml unreg** unregisters the registered kernel modules specified by *id*. Both commands accept a list of module identifiers as arguments. If a registered module has been loaded into memory after its first open, it must be unloaded before it can be unregistered.

ml debug can be used to turn verbose debugging messages on or off or to disable the loading and registering of modules:

−v Turn verbose debugging on.

−n Disable loading and registering of modules.

–s Silence verbose debugging and allow loading and registering of modules.

WARNINGS

A loaded module has all of the system privileges of kernel mode execution.

EXAMPLES

List all loaded and registered modules:

```
ml
```

Load a streams driver with prefix **sdrv** and major number 13:

```
ml ld -v -f strdrv.o -p sdrv -s 13
```

Register a streams module with prefix **tmod** and module name **testmod**:

```
ml reg -m tmod.o -p tmod -a testmod
```

Register a streams module with prefix **tmod** and default module name **tmod**:

```
ml reg -m tmod.o -p tmod
```

Unload the module with *id* 1015:

```
ml unld 1015
```

Load the kernel debug module:

```
ml ld -i /var/sysgen/boot/idbg.o
```

FAILURES

ml failure codes and descriptions are listed in the header file */usr/include/sys/mload.h*.

SEE ALSO

mload(4).

BUGS

The *ml* command does not provide a way to create edt structures for drivers. Driver initialization can only be done from the driver's **init** and **start** functions. See the *lboot*(1M) reference page for loading drivers with edt functions.

NAME

mount, umount – mount and unmount filesystems

SYNOPSIS

mount
mount [**–M** altmtab] [**–P** prefix] **–p**
mount [**–h** host] [**–fnrv**]
mount **–a**[**cfnv**] [**–t** type] [**–T** list]
mount [**–cfnv**] [**–t** type] [**–T** list] [**–b** list]
mount [**–cfnrv**] [**–t** type] [**–T** list] [**–o** options] fsname dir
mount [**–cfnrv**] [**–o** options] fsname | dir

umount **–a**[**kv**] [**–t** type] [**–T** list]
umount **–h** host [**–kv**] [**–b** list]
umount [**–kv**] fsname | dir [fsname | dir] ...

DESCRIPTION

mount attaches a named filesystem *fsname* to the filesystem hierarchy at the pathname location *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**.

umount unmounts a currently mounted filesystem, which can be specified either as a mounted-on directory or a filesystem.

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/usr /usr efs rw 0 0

then the commands **mount /usr** and **mount /dev/usr** are shorthand for **mount /dev/usr /usr**.

MOUNT OPTIONS

–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 with **–t,** all of the filesystems in */etc/fstab* with that type are mounted. Multiple types may be specified with the **–T** option. Filesystems are not necessarily mounted in the order listed in */etc/fstab*.

–b *list* (all-but) Attempt to mount all of the filesystems listed in */etc/fstab* except for those associated with the directories contained in *list*. *list* consists of one or more directory names separated by commas.

−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 **−y** option.

−f Fake a new */etc/mtab* entry, but do not actually mount any filesystems.

−h *host* Mount all filesystems listed in */etc/fstab* that are remote-mounted from *host*.

−n Mount the filesystem without making an entry in */etc/mtab*.

−o *options* Specify *options,* a list of comma-separated words, described in *fstab*(4).

−p Print the list of mounted filesystems in a format suitable for use in */etc/fstab.*

−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.

−t *type* The next argument is the filesystem type. The accepted types are **proc**, **efs**, **xfs**, **nfs**, **fd**, **cachefs**, **dos**, **hfs** and **iso9660**; see *fstab*(4) for a description of these filesystem types. When this option is used, mount calls another program of the form **mount_typename**, where typename is one of the above types. This program must be on the default path.

−T *list* The next argument is a comma-separated list of filesystem types. This option is usually used in combination with **−a** or **−b**.

−v (verbose) *mount* displays a message indicating the filesystem being mounted and any problems encountered.

−M *altmtab*
Instead of */etc/mtab*, use the mtab or fstab *altmtab*.

−P *prefix* Used with the **−p** option, prepends *prefix* to the emitted *filesystem* and *directory* paths. Doesn't alter pathnames embedded in the options, such as the filesystem's **raw**=*path* raw device pathname.

UMOUNT OPTIONS

−a Attempt to unmount all the filesystems currently mounted (listed in */etc/mtab*). In this case, *fsname* is taken from */etc/mtab*.

−b *list* (all-but) Attempt to unmount all of the filesystems currently mounted except for those associated with the directories contained in *list*. *list* consists of one or more directory names separated by commas. Using

umount -a

itself is not usually a good idea, because it can not be reversed by the command

mount -a

since a number of filesystems are often not in the */etc/fstab* file. Among these are the *proc* and *fd* filesystems. Instead, use a command similar to

umount -T xfs,efs

–h *host* Unmount all filesystems listed in */etc/mtab* that are remote-mounted from *host*.

–k Attempt to kill processes that have open files or current directories in the appropriate filesystems and then unmount them.

–t *type* Unmount all filesystems of a given filesystem type. The accepted types are **proc**, **efs**, **xfs**, **nfs**, **fd**, **dos**, **hfs**, and **iso9660**.

–T *list* Unmount all filesystems whose type is in the comma-separated list given.

–v (verbose) *umount* displays a message indicating the filesystem being unmounted and any problems encountered.

EXAMPLES

```
mount /dev/usr /usr                 mount a local disk
mount –avt efs                      mount all efs filesystems; be verbose
mount –t nfs server:/d /net/d       mount remote filesystem
mount server:/d /net/d              same as above
mount –o soft server:/d /net/d      same as above but soft mount
mount –p > /etc/fstab               save current mount state
mount –t dos /dev/rdsk/fds0d2.3.5 /floppy
                                    mount a MS-DOS floppy
mount –t hfs /dev/rdsk/fds0d3.3.5hi /floppy
                                    mount a Macintosh HFS floppy
mount –t iso9660 /dev/scsi/sc0d7l0 /cdrom
                                    mount an ISO 9660 CD-ROM
mount server:/cdrom /net/cdrommount remote iso9660 filesystem
mount –M /root/etc/fstab –P /root –p |
  sed 's;raw=/;raw=/root/' >> /etc/fstab
                                    append /root/etc/fstab with /root
                                    prefix to currently active fstab.
umount –t nfs –b /foo               unmount all nfs filesystems except /foo
```

ERROR MESSAGES

From *mount*:

`mount: device on` *mountdir*: Invalid argument

This message appears for a wide variety of problems. It doesn't usually indicate that you have specified the command line incorrectly; rather that there is something wrong with the disk partition, the filesystem in the disk partition, or the mount directory. For example, this error message occurs if you try to mount a device that doesn't contain a valid filesystem.

From *umount*:

mountdir: Resource busy

Possible causes of a this message are: open files in the filesystem, programs being executed from the filesystems, and users whose current directory is in the filesystem.

Usually it is not possible to unmount the */usr* filesystem because many daemons, such as */usr/lib/lpsched*, */usr/etc/ypbind*, and */usr/etc/syslogd*, execute from the */usr* filesystem. The simplest way to make sure the */usr* filesystem is not busy is to bring the system down to single-user mode with the *single*(1M) command.

You can force all filesystems except the root filesystem to be unmounted with the *umount* **–k** option (note that this kills processes). To unmount the root filesystem, you must be running the miniroot.

FILES

/etc/fstab	filesystem table
/etc/mtab	mount table

SEE ALSO

fsck(1M), mountd(1M), nfsd(1M), mount(2), umount(2), fstab(4), mtab(4).

BUGS

umount can mismanage the */etc/mtab* mount table if another *mount* or *umount* call is in progress at the same time.

Mount calls another "helper" program of the form **mount_typename**, where typename is one of the accepted mount types. If this program is not on the default path, then mount returns with an error message about unknown filesystem. The user must make sure that the helper mount program is in the path. For example, */usr/etc* must be in the path to mount an iso9660 CD.

NOTE

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.

The helper program **mount_iso9660** is in the optional package *eoe.sw.cdrom*. This package must be installed in order to mount iso9660 filesystems.

NAME

network – network initialization and shutdown script

SYNOPSIS

/etc/init.d/network [**start** | **stop**]

DESCRIPTION

The *network* shell script is called during system startup from */etc/rc2* to initialize the standard and optional network devices and daemons. The script is called during system shutdown from */etc/rc0* to gracefully kill the daemons and inactivate the devices.

When called with the *start* argument, the *network* script does the following, using the various configuration flags described below:

- Defines the hostname and hostid based on the name in */etc/sys_id* and its corresponding Internet address in */etc/hosts*.
- Checks that the host's Internet address is not the default 192.0.2.1 Internet test address. If the address is the default address, the software is configured for standalone mode. An Internet address other than the default must be chosen in order to configure the network properly. See the guide *IRIX Admin: Networking and Mail* for information on selecting an address.
- Initializes the network interfaces. The HyperNet interface is initialized if the hypernet configuration flag is on. If multiple ethernet or FDDI interfaces are present, the script computes typical primary and gateway interface names and addresses for most systems. */etc/config/netif.options* provides a place for site-dependent network interface definitions. You need to modify this file only if:
  - the computed primary and/or gateway interface names are incorrect
  - you don't like convention used to define addresses for interfaces
  - the host has more than 2 ethernet or FDDI interfaces

  Each interface must have a unique Internet address and hostname in */etc/hosts*. The script derives the names from */etc/sys_id*. The prefix **gate–** is prepended to the hostname to generate the second interface's name. The suffix **–hy** is appended to generate the HyperNet interface's name. For example:

```
191.50.1.7     yosemite.parks.us       yosemite
137.254.2.49   gate-yosemite.parks.us  gate-yosemite
191.51.0.88    yosemite-hy.parks.us    yosemite-hy
```

  See the comments in */etc/config/netif.options* for details.

- Deletes existing routes.
- Starts the standard networking daemons such as the routing, portmap and DNS nameserver daemons. Initializes the default multicast route.
- (If NFS option is installed). Defines the NIS domain name using */var/yp/ypdomain* if it exists. If the NIS domain is the same as the Internet domain name in */etc/sys_id*, then *ypdomain* is not needed. Starts NIS daemons, mounts and exports NFS filesystems, starts NFS automount, lock and status daemons.
- Starts the *inetd*, *timed*, *timeslave*, *rarpd*, and *rwhod* daemons.
- Starts the 4DDN software (if installed).

When called with the *stop* argument, the *network* script gracefully terminates daemons in the correct order, unmounts NFS filesystems and inactivates the network interfaces.

CONFIGURATION FLAGS

A daemon or subsystem is enabled if its configuration flag in the */etc/config* directory in the **on** state. If a flag file is missing, the flag is considered **off**. Use the *chkconfig*(1M) command to turn a flag **on** or **off**. For example,

```
chkconfig timed on
```

enables the timed flag. When invoked without arguments, *chkconfig* prints the state of all known flags.

There are two special flags: verbose and network. The verbose flag controls the printing of the names of daemons as they are started and the printing of NFS-mounted filesystem names as they are mounted and unmounted. The network flag allows incoming and outgoing traffic. This flag can be set off if you need to isolate the machine from network without removing cables.

The following table lists the configuration flags used to initialize standard and optional software.

Flag	Action if **on**
gated	Start Cornell Internet super-routing daemon
mrouted	Start Stanford IP multicast routing daemon
named	Start 4.3BSD Internet domain name server
rtnetd	Initialize preemptable networking for real-time use
rwhod	Start 4.3BSD *rwho* daemon
timed	Start 4.3BSD time synchronization daemon
timeslave	Start SGI time synchronization daemon
hypernet	Initialize HyperNet controller and routes
nfs	Start NFS daemons, mount NFS filesystems
automount	Start NFS automounter daemon
lockd	Start NFS lock and status daemons
rarpd	Start the Reverse ARP daemon

yp	Enable NIS, start *ypbind* daemon
ypserv	If **yp** is **on**, become a NIS server
ypmaster	If **yp** is **on**, become the NIS master; start password server; **ypserv** should be **on**, too
4DDN	Initialize 4DDN (DECnet connectivity) software

Site-dependent options for daemons belong in ''options'' files in */etc/config*. Certain daemons require options so their options file must contain valid information. See the guide *IRIX Admin: Networking and Mail* and the daemon's manual page in section 1M for details on valid options.

File	Status	
automount.options	optional	
biod.options	optional	
gated.options	optional	
ifconfig–1.options	optional	(for primary network interface)
ifconfig–2.options	optional	(for gateway network interface)
ifconfig–3.options	optional	(for 2nd gateway network interface)
ifconfig–4.options	optional	(for 3rd gateway network interface)
ifconfig–hy.options	optional	(for HyperNet interface)
inetd.options	optional	
mrouted.options	optional	
named.options	optional	
netif.options	optional	(to select different primary & gateway interfaces, etc.)
nfsd.options	optional	
portmap.options	optional	
rarpd.options	optional	
routed.options	optional	
rpc.passwd.options	optional	
rwhod.options	optional	
timed.options	optional	
timeslave.options	required	
ypbind.options	optional	
ypserv.options	optional	

Site-dependent configuration commands to start and stop local daemons, add static routes and publish arp entries should be put in a separate shell script called */etc/init.d/network.local*. Make symbolic links in */etc/rc0.d* and */etc/rc2.d* to this file to have it called during system startup and shutdown:

```
ln –s /etc/init.d/network.local /etc/rc0.d/K39network
ln –s /etc/init.d/network.local /etc/rc2.d/S31network
```

See */etc/init.d/network* for the general format of the script.

FILES

/etc/init.d/network	
/etc/rc0.d/K40network	linked to network
/etc/rc2.d/S30network	linked to network
/etc/config	configuration flags and options files
/etc/sys_id	hostname
/etc/hosts	Internet address-name database
/var/yp/ypdomain	NIS domain name

SEE ALSO

chkconfig(1M), rc0(1M), rc2(1M).

IRIX Admin: Networking and Mail

NAME

nvram, sgikopt – get or set non-volatile RAM variables

SYNOPSIS

nvram [**–v**] [*name* [*value*]]
sgikopt [*name*...]

DESCRIPTION

nvram can be used to set or print the values of non-volatile RAM variables.

When invoked with no arguments, *nvram* displays all known variables in the *name=value* form. The *nvram* arguments are:

name Print the value of *name*.

value If *name* is defined in non-volatile RAM, replace *name*'s definition string with *value*.

–v Print a line of the form *name=value* after getting or setting the named variable.

If invoked as *sgikopt*, more than one *name* can be given. *name*s that do not match known variables are ignored.

The exit status is 1 if any arguments do not match and 0 otherwise.

NOTES

Non-volatile RAM contains a small set of well-known strings at fixed offsets. *nvram* can not be used to define new variables.

Only the superuser can set variables.

The term "non-volatile RAM" is somewhat misleading, because some variables are placed only in volatile RAM and are reset on power-up. Different systems have different mixes of volatile and non-volatile variables.

DIAGNOSTICS

If an attempt to get or set a variable fails for any reason, *nvram* prints an appropriate message on standard error and exits with non-zero status.

Not all systems support the ability to change the contents of non-volatile memory with the *nvram* command. To change the contents of non-volatile memory on systems that do not support *nvram*, you must use the PROM monitor *setenv* command.

SEE ALSO

prom(1M), sgikopt(2), syssgi(2).

NAME

prom – PROM monitor

DESCRIPTION

The PROM monitor is a program that resides in permanently programmed read-only memory, which controls the startup of the system. The PROM is started whenever the system is first powered on, reset with the reset button, or shutdown by the administrator.

The PROM contains features that vary from system to system. Description of various commands, options, and interfaces below may not apply to the PROM in your system and may vary between systems. Furthermore, because PROMs are not normally changed after the manufacture of the system, newly added features are not present in older systems.

Some systems, such as the Indigo R4000, Indigo^2, Indy, Onyx, and CHALLENGE contain an ARCS PROM. Machines that contain an MIPS R8000 such as the POWER CHALLENGE, POWER Onyx and the POWER Indigo^2 use a 64-bit version of the ARCS PROM. The ARCS PROM offers the same functionality as previous PROMs, but in some cases with a different interface. Refer to the ARCS PROM section below for details.

When the system is first powered on, the PROM runs a series of tests on the core components of the system. It then performs certain hardware initialization functions such as starting up SCSI hard disks, initializing graphics hardware and clearing memory. Upon successful completion of these tasks, the PROM indirectly starts the operating system by invoking a bootstrap loader program called *sash*, which in turn reads the IRIX kernel from disk and transfers control to it.

Menu Commands

By default, the PROM attempts to boot the operating system kernel when the system is powered on or reset. Before doing so, however, the opportunity to press the <Escape> key is given. If the <Escape> key is pressed within approximately ten seconds, the PROM displays a menu of alternate boot up options. These other choices allow various types of system maintenance to be performed:

1. Start System

This option causes the system to boot in the default way. It is the same as if the system had been allowed to boot on its own.

2. Install System Software

This option is used when system software needs to be installed or upgraded. The PROM first attempts to find a tape drive on the system and if one is found, it prompts the user to insert the installation tape in it. If a tape device is not found, then installation is expected to take place by Ethernet. In this case, the PROM prompts the user for the name of the system that will be used as the server.

Systems with an ARCS PROM uses a menu to select the installation device. See ARCS PROM section below for details.

3. Run Diagnostics
This option invokes the extended hardware diagnostic program, which performs a thorough test of the CPU board and any graphics boards present. It reports a summary.

4. Recover System
This option can be used to perform special system administration tasks such as restoring a system disk from backup tapes. It follows a sequence similar to installing system software, but instead of starting the installation program, it invokes an interactive restoration tool.

5. Enter Command Monitor
Additional functions can be performed from an interactive command monitor. This option puts the PROM into a manual mode of operation.

6. Select Keyboard Layout
Some systems display a sixth option when the console is on the graphics display which allows the keyboard map to be interactively selected for SGI supported international keyboards.

Manual Mode

The PROM command monitor allows the user to customize certain features of the boot process for one-time only needs or longer term changes. The command monitor has some features that are similar to an IRIX shell such as command line options and environment variables. Some of the environment variables used in the PROM are stored in nonvolatile RAM, which means that their values are preserved even after the power to the system is turned off.

The command monitor has a different method of specifying disks and files than is used under IRIX. A pathname is formed by prefixing the filename with a device name as shown:

devicename(*controller*,*unit*,*partition*)*filename*

Valid device names include:

tpsc SCSI tape drive
dksc SCSI disk drive
bootp network by BOOTP and TFTP protocols (ethernet only)

The *controller* designates which hardware controller to use if multiple controllers for the same type of device exist. Controllers are numbered starting at zero. The *unit* designates which drive to use when a single controller is used with multiple drives. When used with a SCSI device, the unit number is the same as the SCSI target number for the drive. The *partition* designates which disk partition is to be used. Partitions are numbered 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. The controller, unit, and partion all default to zero.

The devices supported by the PROM varies from system to system.

Manual Mode Commands

auto Attempts to boot the system into normal operation. This is the equivalent of the **Start System** menu command.

boot [**–f**] [**–n**] *pathname*
Starts an arbitrary standalone program or kernel as specified by its arguments. The **–f** option suppresses the invocation of the bootstrap loader program. The **–n** option causes the named program to be loaded, but not started.

eaddr Prints the Ethernet address of the built-in Ethernet controller. This address is set at the factory and cannot be changed.

date [*mmddhhmm*[*ccyy*|*yy*][**.***ss*]]
Prints the date or sets the date when given an argument. The PROM does not understand time zones, so times should be given relative to GMT.

exit Exits manual mode and returns to the PROM menu.

help Displays a short summary of the commands available in manual mode.

init Causes a partial restart of the PROM. This command can be used to change the default console immediately. See the **console** environment variable.

hinv Lists the hardware present in the system. This list includes any disk or tape drives, memory, and graphics options. It lists only those devices known to the PROM and may not include all optional boards.

ls *device* List files contained on the device specified. This can be used to examine devices whose layout is known by the PROM such as the disk volume header. It cannot be used to list directories on disk partitions containing IRIX filesystems.

off Turns off the power. Supported only on a subset of systems with software power control.

passwd Set the PROM password. The PROM password can be set to restrict operation of certain PROM modes. With a password set, any attempt to do anything other than a standard system boot requires that the password be reentered. The password is remembered after the system is powered off.

If the password is forgotten, some systems allow the superuser to reset it while running IRIX. Use the *nvram* command to set the **passwd_key** variable to a null string (**nvram passwd_key ""**). Other systems also have a jumper on the system board that can be removed to disable the PROM password. In addition some systems force the **console** environment variable to **g** while the jumper is removed. This jumper should only be removed temporarily in order to reset the

password or fix the **console** environment variable. Indy and Indigo^2 have this feature.

printenv List the current state of the PROM environment variables. Some of the variables listed retain their value after the system is powered off.

resetenv Set all of the PROM nonvolatile environment variables to their factory defaults. This does not affect the PROM password.

resetpw Remove the PROM password. With no PROM password set, all commands and menu options function without restriction.

setenv [−p] *variable value*
Set the specified environment variable to a particular value. Environment variables that are stored in nonvolatile RAM are changed there as well. The **−p** option specifies that this variable should be saved as a *persistent* variable by means of adding the variable to nonvolatile RAM. This is particularly useful for setting frequently used options when starting up the system. Note that a fixed nonvolatile RAM variable is not superseded by this option, but the command behaves as if the **−p** flag is not present. Currently this option is available only on the Indy.

single Start the system in single user mode. The system is booted as in the **auto** command described above, except that it enters initstate **s** instead of initstate **2**. See *init*(1M) for more information on initialization states.

unsetenv *variable*
Disassociates any value with the named environment variable.

version Prints a message containing information about the PROM.

In addition to the commands above, a pathname can be entered directly, which the PROM attempts to load and execute.

PROM Environment Variables

netaddr Used when booting or installing software from a remote system by Ethernet. This variable should be set to contain the Internet address of the system. It is stored in nonvolatile RAM.

dbaud Diagnostic baud rate. It can be used to specify a baud rate other than the default when a terminal connected to serial port #1 is to be used as the console. This variable is stored in nonvolatile RAM.

bootfile This variable controls two aspects of the automatic boot up process. First, it names the standalone loader that is used as an intermediary when booting from disk. Second, the device portion of the filename is used to determine the default boot disk. The PROM assumes that the disk specified as part of the standalone loader pathname is the disk where the IRIX root filesystem exists. Furthermore, during software installation, the PROM uses that disk's swap partition for the miniroot. The actual partitions assumed by the PROM to contain the root

filesystem and swap area are determined by reading the volume header. See *vh*(7M) for more information. This variable is stored in nonvolatile RAM.

bootmode The default mode of operation after you turn on power to the system is determined by the **bootmode** variable. If the **bootmode** is set to **c**, then the system is automatically booted whenever it is reset or power is turned on to the system. If the **bootmode** is set to **m**, the PROM displays the menu and waits for a command instead. Setting **bootmode** to **d** has the same affect as **m**, with the addition of more verbose power-on diagnostics. This variable is stored in nonvolatile RAM.

boottune Selects among the available boot tunes It is specified as a small integer such as **1**, which is the default tune. A setting of **0** selects a random tune. Currently only the POWER Indigo^2 supports this variable. This variable is stored in nonvolatile RAM.

autopower
On systems with software power control, a setting of **y** allows the system to automatically power back on after an AC power failure. The default setting of **n** requires the power switch to be pressed to restart the system. This variable is stored in nonvolatile RAM.

console The system console can be set with the **console** variable. If **console** is set to **g** or **G**, the console is assumed to be the graphics display. On some systems with multiple graphics adapters, setting **console** to **g0** (identical to **g**), **g1**, or **g2** can be used to select alternate graphics displays. If **console** is set to **d**, the console is assumed to be a terminal connected to the first serial port. In addition, some systems also accept **d2** for a terminal connected to second serial port. Lastly, this can be overridden on some systems by removing the password jumper and forcing the console to **g**, which is useful for for recovering from setting **console** to **d** when a terminal is not available. This variable is stored in nonvolatile RAM.

diskless If set to **1**, the kernel assumes that the system is to be started up as a diskless node. This variable is stored in nonvolatile RAM.

monitor Overrides the default monitor setting when an unrecognized monitor is attached to an Indy system. Specifying **h** or **H** indicates the attached monitor supports high resolution mode (1280x1024 @ 60Hz). Otherwise the default resolution is low resolution (1024x768 @ 60Hz). This variable is usable only on an Indy system and is stored in nonvolatile RAM.

nogfxkbd If set to **1**, the system does not require the keyboard to be plugged in. By default, if the console is the graphics display and the keyboard is not plugged in or is otherwise unresponsive to commands, it is assumed to be broken. The system switches to the serial terminal console and waits for a command. This variable is stored in nonvolatile RAM.

notape If set to **1**, the PROM assumes that the Ethernet is to be used for software installation or system recovery even if a tape drive is present on the system. By default, if the PROM sees a tape drive in the hardware inventory, it assumes that it will be used for software installation; setting **notape** allows that assumption to be overruled.

volume Sets the speaker volume during boot up. This controls the volume of the startup, shutdown, and bad graphics tunes generated on systems with integral audio hardware. This variable is stored in nonvolatile RAM.

pagecolor Sets the background color of the textport set with a six character string of hex RGB values. This variable is stored in nonvolatile RAM.

path The **path** variable is used with some commands to provide a default device name. It is derived from the **bootfile** variable.

prompoweroff
If set to **y**, the IRIX operating system returns to the PROM to do the actual powering off of the system. Powering off the system by the PROM is preceded by the playing of the "shutdown" tune that is normally played when returning to the PROM monitor via the *shutdown* or *halt* commands. This variable is available only on Indy systems and must be set with the command **setenv −p prompoweroff y** command to retain the setting after power is turned off.

rebound If set to **y**, the system attempts to automatically reboot in the event of a kernel panic overriding the value of the **reboot_on_panic** systune parameter. This variable is stored in nonvolatile RAM.

sgilogo If set to **y**, the SGI logo and other product information is shown on systems that support the standalone GUI. This variable is stored in nonvolatile RAM.

ARCS PROM

Machines with the ARCS PROM behave similar to what is described above. Changes were made to support the Advanced Computing Environment's (ACE) Advanced Risc Computing Standard (ARCS), provide a graphical user interface, and clean up various loopholes in older PROMs. In many cases efforts were made to maintain old syntax and conventions.

The ARCS document describes system requirements, which includes minimum system function, procedure entry points, environment variables, hardware inventory, and other system conventions. Programmatic interfaces and other hardware requirements are outside the scope of this reference page.

ARCS pathnames are tied directly to the hardware inventory, which is stored in a tree that represents the system's device architecture. It is rooted with a system entry and grows to peripheral devices such as a disk drive. ARCS pathnames are written as a series of *type(unit)* components that parallel the inventory tree.

Old-style pathnames are automatically converted to new-style pathnames, so the old names can still be used. The PROM matches the first device described by the pathname, so full pathnames are not always required. The **–p** option to **hinv** prints the pathnames to all user accessible devices. Some examples of common pathnames are:

scsi(0)disk(1)partition(1)	`dksc(0,1,1)`
disk(1)part(1)	`same as above`
scsi(0)cdrom(5)partition(7)	`dksc(0,5,7)`
network(0)bootp()host:file	`bootp()host:file`
serial(0)	`first serial port`
keyboard()	`graphics keyboard`
video()	`graphics display`

ARCS defines environment variables that provide the same function as in older PROMs, but with different names and values:

ConsoleIn/ConsoleOut
These two variables are set at system startup automatically from the **console** variable. They are maintained only for ARCS compatibility only.

OSLoadPartition
The device partition where the core operating system is found. For IRIX, this variable is used as the root partition when the **root** variable is unused and the device configured in the kernel variable **rootdev** is not available. This variable is stored in nonvolatile RAM, but is normally left unset, which allows the PROM to automatically configure it at system power-on.

OSLoader The operating system loader. For IRIX, this is *sash*. This variable is stored in nonvolatile RAM, but is normally left unset, which allows the PROM to automatically configure it at system power-on.

SystemPartition
The device where the operating system loader is found. This variable is stored in nonvolatile RAM, but is normally left unset, which allows the PROM to automatically configure it at system power-on.

OSLoadFilename
The filename of the operating system kernel. For IRIX this is */unix*. This variable is stored in nonvolatile RAM, but is normally left unset, which allows the PROM to automatically configure it at system power-on.

OSLoadOptions
The contents of this variable are appended to the **boot** command constructed when autobooting the system. This variable is stored in nonvolatile RAM.

AutoLoad Controls if the system boots automatically on reset/power cycle. Can be set to **Yes** or **No**. Previously this function was controlled by setting **bootmode** to **c** or **m**. This variable is stored in nonvolatile RAM.

To try and improve the looks and usability of the PROM, the ARCS PROM uses a graphical interface when **console=g**. In all cases the keyboard can be used instead of the mouse, and in most cases the familiar keystrokes from previous PROMs work.

For example, the traditional five item menu consists of a list of buttons containing one icon each. To make a selection, either click any mouse button with the button, or press the corresponding 1 through 5 key.

The only major user interface changes are for **Install Software** and **Recover System** (menu items 2 and 4). The interface allows interactive selection of a device type and then selection among devices of that type. This makes it easier than previous PROMS to install from local drives or remote directories without hacks like **notape** and **tapedevice**.

The set of commands available from the command monitor is relatively unchanged:

hinv By default **hinv** prints a formatted abbreviated list similar to the old-style PROM. A **–t** option has been added to print the ARCS configuration tree directly. A secondary option **–p**, valid only with **–t**, prints the corresponding ARCS pathnames for peripheral devices.

There has also been some changes/additions to the SGI-defined environment variables:

diskless This controls if the system is run as a diskless system. Since some of the other environment variables are changed for ARCS compliance, diskless setup is slightly different. The environment should be set as follows.

```
diskless=1
SystemPartition=bootp()host:/path
OSLoader=kernelname
```

keybd Normally this variable is left unset and the system automatically configures the keyboard to use its native key map. To override the default, **keybd** should be set to a three to five character string. The following strings are recognized, depending on the PROM revision: USA, DEU, FRA, ITA, DNK, ESP, CHE-D, SWE, FIN, GBR, BEL, NOR, PRT, CHE-F or US, DE, FR, IT, DK, ES, de_CH, SE, FI, GB, BE, NO, PT, fr_CH on systems with the keyboard layout selector. On newer systems, JP is also acceptable.

Alternatively you can select between swiss french and swiss german by setting **keybd** to **d** or **D** for the german map. On systems with PC keyboards, a string not matching one of the above is passed to the X server and used as the name of the keyboard map to load. This variable is stored in nonvolatile RAM.

diagmode If set to **v**, power-on diagnostics are verbose. In addition, more diagnostics are run. This is similar to **bootmode=d**, however it does not affect the behavior of AutoLoad. This variable is stored in nonvolatile RAM.

The ARCS standard specifies different error numbers than IRIX:

```
ESUCCESS            0
E2BIG               1
EACCES              2
EAGAIN              3
EBADF               4
EBUSY               5
EFAULT              6
EINVAL              7
EIO                 8
EISDIR              9
EMFILE              10
EMLINK              11
ENAMETOOLONG        12
ENODEV              13
ENOENT              14
ENOEXEC             15
ENOMEM              16
ENOSPC              17
ENOTDIR             18
ENOTTY              19
ENXIO               20
EROFS               21
EADDRNOTAVAIL       31
ETIMEDOUT           32
ECONNABORTED        33
ENOCONNECT          34
```

Examples

To boot the disk formatter, *fx*(1M), from a local tape containing the installation tools:

1. Get into the command monitor by choosing option **5** from the menu.

2. Determine the type of CPU board in your system with the *hinv* command. The board type is listed as the letters **IP** followed by a number. Also, look for the item that lists the tape drive to determine the format of the device name. For instance, a SCSI tape addressed as device 7 might be listed as **SCSI tape: tpsc(0,7)** in which case the device is **tpsc(0,7)**.

3. With the installation tools tape in the drive, boot *fx* as follows:

```
boot -f tpsc(0,7)fx.IP6
```

where **tpsc(0,7)** is the device name and **IP6** is the CPU board type.

To change the system console from the graphics display to a terminal connected to serial port #1:

1. Get into the command monitor by choosing option **5** from the menu.
2. Change the **console** variable to **d** as follows:

```
setenv console d
```

3. Reinitialize the PROM with the **init** command:

```
init
```

SEE ALSO

bootp(1M), fx(1M), inst(1M), nvram(1M), tftpd(1M).

NAME

prtvtoc – print disk volume header information

SYNOPSIS

/etc/prtvtoc [[–**aefhms**] [–**t** *fstab*]] [*rawdiskname*]

DESCRIPTION

prtvtoc prints a summary of the information in the volume header for a single disk or all of the local disks attached to a system (see **vh**(7M)). The command is usually used only by the superuser.

The *rawdiskname* name should be the raw device filename of a disk volume header in the form */dev/rdsk/xxs?d?vh.*

Note: *prtvtoc* knows about the special file directory naming conventions, so the */dev/rdsk* prefix can be omitted.

If no name is given, the information for the root disk is printed.

In single disk mode, *prtvtoc* prints information about the disk geometry (number of cylinders, heads, and so on), followed by information about the partitions. For each partition, the type is indicated (for example, filesystem, raw data, and so on). Cylinders can be non-integral values, as they may not correspond to actual physical values, for some drive types. For filesystem partitions, *prtvtoc* shows if there is actually a filesystem on the partition, and if it is mounted, the mount point is shown. Mount points shown in square brackets indicate the mount point of the logical volume the partition belongs to.

The following options to *prtvtoc* can be used:

–**s** Print only the partition table, with headings but without the comments.

–**h** Print only the partition table, without headings and comments. Use this option when the output of the *prtvtoc* command is piped into another command.

–**t** *fstab* Use the file *fstab* instead of */etc/fstab*.

The following options create summaries from all disk volume headers:

–**a** Show abbreviated partition listings for all disks attached to the system.

–**m** List all partitions in use by local filesystems. The listing includes partitions that belong to logical volumes.

–**e** Extended listing. This combines the **-a** and **-m** options as well as reporting unallocated (free) partitions, and overlapping mounted partitions.

EXAMPLE

The output below is for a SCSI system (root) disk obtained by invoking *prtvtoc* without parameters.

```
Printing label for root disk

* /dev/rdsk/dks0d1vh (bootfile "/unix")
*     512 bytes/sector
*      74 sectors/track
*      15 tracks/cylinder
*       3 spare blocks/cylinder
*    1876 cylinders
*       3 cylinders occupied by header
*    1873 accessible cylinders
*
* No space unallocated to partitions

Partition Type  Fs   Start: sec  (cyl)  Size: sec    (cyl)  Mount
Directory
 0         xfs  yes      3321   (   3)      32103   (  29)   /
 1         raw          35424   (  32)      81918   (  74)
 6         xlv         117342   ( 106)    1959390   (1770)   [/usr]
 7         efs           3321   (   3)    2073411   (1873)
 8      volhdr              0   (   0)       3321   (   3)
10      volume              0   (   0)    2076732   (1876)
```

This next output is for a SCSI option disk obtained by invoking *prtvtoc* with drive dks0d2vh as the parameter.

```
* /dev/rdsk/dks0d2vh (bootfile "/unix")
*     512 bytes/sector
*      85 sectors/track
*       9 tracks/cylinder
*       3 spare blocks/cylinder
*    2726 cylinders
*       4 cylinders occupied by header
*    2722 accessible cylinders
*
* No space unallocated to partitions

Partition  Type  Fs   Start: sec   (cyl)    Size: sec    (cyl)  Mount
Directory
 7          xfs  yes        3048  (   4)      2065782   (2711)   /work
 8       volhdr                0  (   0)         3048   (   4)
```

```
        10       volume                0  (   0)      2077212   (2726)
        15       xfslog          2068830  (2715)         8382   (  11)
```

SEE ALSO

dvhtool(1M), fx(1M), dks(7M), usraid(7M), vh(7M).

NAME

ps – report process status

SYNOPSIS

ps [options]

DESCRIPTION

ps prints information about active processes. Without *options*, information is printed about processes associated with the controlling terminal. The output consists of a short listing containing only the process ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the selection of *options*.

options accept names or lists as arguments. Arguments can be either separated from one another by commas or enclosed in double quotes and separated from one another by commas or spaces. Values for *proclist* and *grplist* must be numeric.

The *options* are:

–a Print information about **a**ll processes most frequently requested: all those except process group leaders and processes not associated with a terminal.

–A Print information about **e**very process now running.

–c Print information about the scheduler properties. (See below.)

–d Print information about all processes except process group leaders.

–e Print information about **e**very process now running (equivalent to **–A**).

–f Generate a **f**ull listing. (See below for significance of columns in a full listing.)

–g *grplist* List only process data whose process group leader's ID numbers appear in *grplist*. (A group leader is a process whose process ID number is identical to its process group ID number. A login shell is a common example of a process group leader.)

–G *grplist* List only process data whose real group leader's ID numbers appears in *grplist*.

–j Print session ID and process group ID.

–l Generate a **l**ong listing. (See below.)

–M If the system supports Mandatory Access Control, print the security label for each process. The **–M** option can be automatically be turned on by using an environmental variable LABELFLAG. Set variable to **on** (not case sensitive) for automatic security label information. To turn off feature set to **off** or **NULL**.

–n *name* This argument is obsolete and is no longer used.

–o *format* Print information according to the format specification given in *format*. (See below.)

–p *proclist* List only process data whose process ID numbers are given in *proclist*.

–s *sesslist* List information on all session leaders whose IDs appear in *sesslist*.

–t *termlist* List only process data associated with the terminal given in *termlist*. Terminal identifiers consist of the device's name (for example, **ttyd1**, **ttyq1**).

–u *uidlist* List only process data whose user ID number or login name is given in *uidlist*. In the listing, the numerical user ID is printed unless you give the **–f** option, which prints the login name.

–U *uidlist* List only process data whose read user ID number or login name is given in *uidlist*.

Under the **–f** option, *ps* tries to determine the command name and arguments given when the process was created by examining the user block. Failing this, the command name is printed, as it would have appeared without the **–f** option, in square brackets.

The column headings and the meaning of the columns in a *ps* listing are given below. The letters **f** and **l** indicate the option (**f**ull or **l**ong, respectively) that causes the corresponding heading to appear (assuming the **–o** option is not specified); **all** means that the heading always appears. Note that these options determine only what information is provided for a process; they do not determine which processes are listed.

If the environment variable _XPG is defined and has a numeric value greater than 0, *ps* operates in conformance with the X/Open XPG4 specifications. The format of the output of the **–l** option differs in some details from the XPG format and backward compatibility mode. The differences are explained in the table below.

F (l) Flags (hexadecimal and additive) associated with the process:

001 Process is a system (resident) process.
002 Process is being traced.
004 Stopped process has been given to parent via *wait*(2).
008 Process is sleeping at a non-interruptible priority.
010 Process is in core.
020 Process user area is in core.
040 Process has enabled atomic operator emulation.
080 Process in stream poll or select.
100 Process is a kernel thread.

S (l) The state of the process:

0 Process is running on a processor.
S Process is sleeping, waiting for a resource.
R Process is running.
Z Process is terminated and parent not waiting (*wait*(2)).
T Process is stopped.
I Process is in intermediate state of creation.
X Process is waiting for memory.

UID (f,l) The user ID number of the process owner (the login name is printed under the **−f** option).

PID (all) The process ID of the process (this datum is necessary in order to kill a process).

PPID (f,l) The process ID of the parent process.

PGID (j) Process group leader ID. This can be used with the **−g** option.

SID (j) Session ID. This can be used with the **−s** option.

CLS (c) Scheduling class. The values printed for **CLS** are the two character mnemonics for the scheduler queues displayed by the **−q** option of *pset*(1M).

C (f,l) Processor utilization for scheduling. Not printed when the **−c** option is used.

PRI (l) The priority of the process (higher numbers mean lower priority).

NI (l) Nice value, used in priority computation. Not printed when the **−c** option is used (see *nice*(1) and *csh*(1)). Only processes in the time-sharing class have a nice value. Processes in other scheduling classes have their two letter class mnemonic printed in this field (refer to *schedctl*(2) and *pset*(1M) for information about other scheduling classes).

P (l) If the process is running, gives the number of processor on which the process is executing. Contains an asterisk otherwise. This is not displayed in X/OPEN XPG4 conformance mode.

ADDR (l) The physical address of the process. This is only displayed in X/OPEN XPG4 conformance mode.

SZ (l) Total size (in pages) of the process, including code, data, shared memory, mapped files, shared libraries and stack. Pages associated with mapped devices are not counted. (Refer to *sysconf*(1) or *sysconf*(3C) for information on determining the page size.)

RSS (l) Total resident size (in pages) of process. This includes only those pages of the process that are physically resident in memory. Mapped devices (such as graphics) are not included. Shared memory (*shmget*(2)) and the shared parts of a forked child (code, shared objects, and files mapped **MAP_SHARED**) have the number of pages prorated by the number of processes sharing the page. Two independent processes that use the same shared objects and/or the same code each count all valid resident pages as part of their own resident size. The page size can either be 4096 or 16384 bytes as determined by the return value of the *getpagesize*(2) system call. In general the larger page size is used on systems where *uname*(1) returns "IRIX64". This is not displayed in X/OPEN XPG4 conformance mode.

WCHAN (l) The address of an event for which the process is sleeping, or in SXBRK state, (if blank, the process is running).

STIME (f) The starting time of the process, given in hours, minutes, and seconds. (A process begun more than twenty-four hours before the *ps* inquiry is executed is given in months and days.)

TTY (all) The controlling terminal for the process (the message, **?**, is printed when there is no controlling terminal).

TIME (all) The cumulative execution time for the process.

COMMAND(all)
The command name (the full command name and its arguments are printed under the **−f** option). A process that has exited and has a parent, but has not yet been waited for by the parent, is marked **<defunct>**.

The **−o** option allows the output format to be specified under user control.

The format specification must be a list of names presented as a single argument, blank- or comma-separated. Each variable has a default header. The default header can be overridden by appending an equals sign and the new text of the header. The rest of the characters in the argument are used as the header text. The fields specified are written in the order specified on the command line and should be arranged in columns in the output. The field widths are selected by the system to be at least as wide as the header text (default or overridden value). If the header text is null such as **−o** user=, the field width is at least as wide as the default header text. If all header text fields are null, no header line is written.

The following names are recognized:

ruser The real user ID of the process.
user The effective user ID of the process.
rgroup The real group UD of the process.

group The effective group ID of the process.
pid The decimal value of the process ID.
ppid The decimal value of the parent process ID.
pgid The decimal value of the process group ID.
pcpu The ratio of CPU time used recently to the CPU time available in the same period, expressed as a percentage.
vsz The size of the process in (virtual) memory.
nice The decimal value of the system scheduling priority of the process.
time The cumulative CPU time of the process.
etime The elapsed time since the process was started.
stime The starting time of the process.
flag Flags associated with the process.
state The state of the process.
wchan The event for which the process is waiting or sleeping.
util Processor utilization for scheduling.
uid The user ID number of the process owner.
cpu The processor process is currently executing on.
class The scheduling class of the process.
tty The name of the controlling terminal of the process (if any) in the same format used by the who utility.
comm The name of the command being executed (argv[0] value) as a string.
args The command with all its arguments as a string.

The file */tmp/.ps_data/.ps_data* is used to improve the performance of *ps* by caching uid to username translations, kernel info, and some device information. It is recreated when it is older (either the mtime or ctime) than any of */unix*, */dev*, or */etc/passwd*, or when a read error occurs on the file. *ps* runs noticeably slower when this file isn't used, or needs to be recreated. Note that new NIS users may have jobs reported as numeric values, since the *ps_data* file won't be recreated automatically; removing this file and rerunning *ps* fixes the problem.

SHARE II ACTIONS

When the Share II package is installed and enabled, every process acquires a new property: its *attached lnode*. The *lnode* is the kernel structure that is used to store per-user resource and administration data under Share II. Many processes can be attached to the same lnode.

An lnode contains a user's resource limits, including limits on memory usage and 'number of processes'. All the processes attached to an lnode are collectively subject to the lnode's memory and process limits.

Each lnode is addressed by a unique key which is a UID number. When given the **–y** option, *ps* reports each process's lnode attachment under the 'UID' column as a UID or login name.

FILES

/dev
/dev/tty*
/etc/passwd UID information supplier
/tmp/.ps_data/.ps_data
internal data structure

SEE ALSO

getty(1M), gr_osview(1), gr_top(1), kill(1), nice(1), pset(1M), sysconf(1), top(1), sysconf(3C).

WARNING

Things can change while *ps* is running; the snapshot it gives is only true for a splitsecond, and it may not be accurate by the time you see it. Some data printed for defunct processes is irrelevant.

If no *termlist*, *proclist*, *uidlist*, or *grplist* is specified, *ps* checks *stdin*, *stdout*, and *stderr* in that order, looking for the controlling terminal and attempts to report on processes associated with the controlling terminal. In this situation, if *stdin*, *stdout*, and *stderr* are all redirected, *ps* does not find a controlling terminal, so there is no report.

ps −ef may not report the actual start of a tty login session, but rather an earlier time, when a getty was last respawned on the tty line.

NAME

pwck – password file checker

SYNOPSIS

pwck [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*.

pwck has the ability to parse YP entries in the password file.

FILES

/etc/passwd

SEE ALSO

passwd(4).

DIAGNOSTICS

`Too many/few fields`
An entry in the password file does not have the proper number of fields.

`No login name`
The login name field of an entry is empty.

`Bad character(s) in login name`
The login name in an entry contains one or more non-alphanumeric characters.

`Login name too long`
The login name in an entry has more than 8 characters.

`Invalid UID`
The user ID field in an entry is not numeric or is greater than 65535.

`Invalid GID`
The group ID field in an entry is not numeric or is greater than 65535.

`No login directory`
The login directory field in an entry is empty.

`Login directory not found`
The login directory field in an entry refers to a directory that does not exist.

`Optional shell file not found.`
The login shell field in an entry refers to a program or shell script that does not exist.

`No netgroup name`
The entry is a Yellow Pages entry referring to a netgroup, but no netgroup is present.

`Bad character(s) in netgroup name`
The netgroup name in a Yellow Pages entry contains characters other than lower-case letters and digits.

`First char in netgroup name not lower case alpha`
The netgroup name in a Yellow pages entry does not begin with a lowercase letter.

NAME

pwconv – install and update **/etc/shadow** with information from **/etc/passwd**

SYNOPSIS

pwconv

DESCRIPTION

The **pwconv** command creates and updates **/etc/shadow** with information from **/etc/passwd**.

If the **/etc/shadow** file does not exist, **pwconv** creates **/etc/shadow** with information from **/etc/passwd**. The command populates **/etc/shadow** with the user's login name, password, and password aging information. If password aging information does not exist in **/etc/passwd** for a given user, none is added to **/etc/shadow**. However, the last changed information is always updated.

If the **/etc/shadow** file does exist, the following tasks are performed:

- Entries that are in the **/etc/passwd** file and not in the **/etc/shadow** file are added to the **/etc/shadow** file.
- Entries that are in the **/etc/shadow** file and not in the **/etc/passwd** file are removed from **/etc/shadow**.
- Password attributes (for example, password and aging information) in an **/etc/passwd** entry are moved to the corresponding entry in **/etc/shadow**.

The **pwconv** program is a privileged system command that cannot be executed by ordinary users.

The contents of the **/etc/passwd** and **/etc/shadow** files are saved in **/etc/opasswd** and **/etc/oshadow**, respectively. The system can be restored to its preconversion state by replacing the content of the **/etc/passwd** file with the content of **/etc/opasswd** and removal of **/etc/shadow** (if it did not exist prior to the run of **pwconv**) or its replacement by **/etc/oshadow**. These files are overwritten each time the **pwconv** program is run. The use of some of the system administration tools causes **pwconv** to be run, and therefore the backup files to be overwritten, each time an entry is added, deleted, or modified.

NOTES

There is no NIS equivalent to **/etc/shadow**. Therefore, if a user does not have an entry in **/etc/shadow**, they can not log in. This is because the password field *must* come from **/etc/shadow**. Other NIS information such as user name, home directory, and so on is still available, just not passwords.

pwconv does not copy NIS entries from **/etc/passwd**. In order for such users to be able to log in, entries in **/etc/shadow** must be created by hand on a user by user basis.

FILES

/etc/passwd
/etc/shadow
/etc/opasswd
/etc/oshadow

SEE ALSO

passwd(1).

DIAGNOSTICS

The **pwconv** command exits with one of the following values:

0 Success.
1 Permission denied.
2 Invalid command syntax.
3 Unexpected failure. Conversion not done.
4 Unexpected failure. Password file(s) missing.
5 Password file(s) busy. Try again later.

NAME

rcp – remote file copy

SYNOPSIS

rcp [**−p**] [**−v**] file1 file2
rcp [**−p**] [**−r**] [**−v**] file ... directory

DESCRIPTION

rcp copies files between machines. Each *file* or *directory* argument has one of these forms:

- A local filename, *path*, containing no **:** characters, or a \ before any **:**'s.
- A remote filename of the form *remhost***:***path*.
- A remote filename of the the form *remuser***@***remhost***:***path*, which uses the user name *remuser* rather than the current user name on the remote host.

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

By default, the mode and owner of *file2* are preserved if it already existed; otherwise the mode of the source file modified by the *umask*(2) on the destination host is used.

The options to *rcp* are:

−p Causes *rcp* to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the *umask*.

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

−v Causes the filename to be printed as it is copied to or from a remote host.

rcp does not prompt for passwords; your current local user name must exist on *remhost* and allow remote command execution via *rsh*(1C).

rcp handles third party copies, where neither source nor target files are on the current machine. Hostname-to-address translation of the target host is performed on the source host.

SEE ALSO

cp(1), ftp(1C), rlogin(1C), rsh(1C), hosts(4), rhosts(4).

BUGS

rcp doesn't detect all cases where the target of a copy might be a file in cases where only a directory should be legal.

If you use *csh*(1), *rcp* does not work if your *.cshrc* file on the remote host unconditionally executes interactive or output-generating commands. The message

```
protocol screwup
```

is displayed when this happens. Put the offending commands inside the following conditional block:

```
if ($?prompt) then

endif
```

so they won't interfere with *rcp*, *rsh*, and other non-interactive, *rcmd*(3N)-based programs.

rcp cannot handle filenames that have embedded newline characters. A newline character is a *rcp* protocol delimiter. The error message when this happens is:

```
protocol screwup: unexpected <newline>
```

NAME

Restore – restore the specified file or directory from tape

SYNOPSIS

Restore [*–h hostname*] [*–t tapedevice*] [*directory_name* | *file_name*]

DESCRIPTION

The *Restore* command copies the named file or directory from a local or remote backup tapes to disk. If no file or directory is specified, *Restore* copies all the files found on the tape to disk.

Files are restored into the current directory if the backup tape contains pathnames beginning with "**.**".

Files on disk are overwritten even if they are more recent than the respective files on tape.

The options and arguments to *Restore* are:

–h *hostname* If a tape drive attached to a remote host is used for restoring, specify the name of the remote host with the **–h** *hostname* option. For remote restore to successfully work, you should have a TCP/IP network connection to the remote host and **guest** login privileges on that host.

–t *tapedevice* If the local or remote tape device is pointed to by a device file other than */dev/tape*, the device should be specified by the **–t** *tapedevice* option.

directory_name Restore just the files in the directory *directory_name*.

file_name Restore just the file *file_name*.

The *Restore* command expects the backup tape to be in the special *bru*(1) format written by *Backup*(1) and by the System Manager Backup & Restore tool when doing full (not partial) backups. This is the same format used for system recovery.

SEE ALSO

Backup(1), List_tape(1), bru(1).

NAME

restore, rrestore – incremental filesystem restore

SYNOPSIS

restore key [name ...]
rrestore key [name ...]

DESCRIPTION

restore, and *rrestore* are applicable only to dumps made by *dump*(1m) from EFS filesystems, but they can restore files into any type of filesystem, not just an EFS filesystem.

restore reads tapes dumped with the *dump*(1M) command and restores them *relative to the current directory*. 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. Any arguments supplied for specific options are given as subsequent words on the command line, in the same order as that of the options listed. 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 Restore the entire tape. The tape is read and its full contents loaded into the current directory. This should not be done lightly; the **r** key should only be used to restore a complete level 0 dump tape onto a clear filesystem or to restore an incremental dump tape after a full level 0 restore. Thus

```
/etc/mkfs /dev/dsk/dks0d2s0
/etc/mount /dev/dsk0d2s0 /mnt
cd /mnt
restore r
```

is a typical sequence to restore a complete dump. Another *restore* can be done to get an incremental dump in on top of this. Note that *restore* leaves a file *restoresymtable* in the root directory to pass information between incremental restore passes. This file should be removed when the last incremental tape has been restored. Also, see the note in the BUGS section below.

R Resume restoring. *restore* requests a particular tape of a multi volume set on which to restart a full restore (see the **r** key above). This allows *restore* 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, 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, 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* 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, 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, 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* asks 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* 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 can be used in addition to the letter that selects the function desired.

b The next *argument* to *restore* is used as the block size of the tape (in kilobytes). If the **b** option is not specified, *restore* tries to determine the tape block size dynamically, but is only able to do so if the block size is 32 or less. For larger sizes, the **b** option must be used with *restore*.

f The next *argument* to *restore* is used as the name of the archive instead of */dev/tape*. If the name of the file is –, *restore* reads from standard input. Thus, *dump*(1M) and *restore* can be used in a pipeline to dump and restore a filesystem with the command

```
dump 0f - /usr | (cd /mnt; restore xf -)
```

If the name of the file is of the format *machine*:*device*, the filesystem dump is restored from the specified machine over the network. *restore* creates a remote server */etc/rmt*, on the client machine to access the tape device. Since *restore* is normally run by root, the name of the local machine must appear in the *.rhosts* file of the remote machine. If the filename *argument* is of the form *user@machine:device*, *restore* attempts to execute as the specified use on the remote machine. The specified user must have a *.rhosts* file on the remote machine that allows root from the local machine.

v Normally *restore* 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* does not ask whether it should abort the restore if gets a tape error. It always tries to skip over the bad tape block(s) and continue as best it can.

m *restore* extracts by inode numbers rather than by filename. 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* 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* is a number that selects the dump file when there are multiple dump files on the same tape. File numbering starts at 1.

n Only those files that are newer than the file specified by the next *argument* are considered for restoration. *restore* looks at the modification time of the specified file using the **stat**(2) system call.

e No existing files are overwritten.

E Restores only non-existent files or newer versions (as determined by the file status change time stored in the dump file) of existing files. Note that the *ls*(1) command shows the modification time and not the file status change time. See *stat*(2) for more details.

d Turn on debugging output.

o Normally *restore* does not use *chown*(2) to restore files to the original user and group id unless it is being run by the superuser (or with the effective user id of zero). This is to provide Berkeley-style semantics. This can be overridden with the **o** option which results in *restore* attempting to restore the original ownership to the files.

N Do not write anything to the disk. This option can be used to validate the tapes after a dump. If invoked with the **r** option, *restore* goes through the motion of reading all the dump tapes without actually writing anything to the disk.

DIAGNOSTICS

restore complains about bad key characters.

On getting a read error, *restore* prints out diagnostics. If **y** has been specified, or the user responds **y**, *restore* attempts to continue the restore.

If the dump extends over more than one tape, *restore* asks the user to change tapes. If the **x** or **i** key has been specified, *restore* also asks 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*. Most checks are self-explanatory or can never happen. Common errors are given below.

`Converting to new filesystem format.`
A dump tape created from the old filesystem has been loaded. It is automatically converted to the new filesystem format.

<*filename*>`: not found on tape`
The specified filename 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 filesystem.

`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 filesystem.

`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 filename is specified, its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, 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* may have to resynchronize itself. This message lists the number of blocks that were skipped over.

`Error while writing to file /tmp/rstdir*`
An error was encountered while writing to the temporary file containing information about the directories on tape. Use the TMPDIR environment variable to relocate this file in a directory that has more space available.

`Error while writing to file /tmp/rstdir*`
An error was encountered while writing to the temporary file containing information about the owner, mode and timestamp information of directories. Use the TMPDIR environment variable to relocate this file in a directory that has more space available.

EXAMPLES

```
restore r
```

Restores the entire tape into the current directory, reading from the default tape device */dev/tape*.

```
restore rf guest@kestrel.sgi.com:/dev/tape
```

Restores the entire tape into the current directory, reading from the remote tape device */dev/tape* on host kestrel.sgi.com using the guest account.

```
restore x /etc/hosts /etc/fstab /etc/myfile
```

Restores the three specified files into the current directory, reading from the default tape device */dev/tape*.

```
restore x /dev/dsk
```

Restores the entire */dev/dsk* directory and subdirectories recursively into the current directory, reading from the default tape device */dev/tape*.

```
restore rN
```

Reads the entire tape and go through all the motions of restoring the entire dump, without writing to the disk. This can be used to validate the dump tape.

```
restore xe /usr/dir/foo
```

Restores (recursively) all files in the given directory */usr/dir/foo*. However, no existing files are overwritten.

```
restore xn /usr/dir/bar
```

Restores (recursively) all files that are newer than the given file */usr/dir/bar*.

FILES

/dev/tape	This is the default tape device used unless the environment variable TAPE is set.
/tmp/rstdir*	This temporary file contains the directories on the tape. If the environment variable TMPDIR is set, the file is created in that directory.
/tmp/rstmode*	This temporary file contains the owner, mode, and time stamps for directories. If the environment variable TMPDIR is set, the file is created in that directory.
./restoresymtable	Information is passed between incremental restores in this file.

SEE ALSO

dump(1M), mkfs(1M), mount(1M), rmt(1M), rhosts(4), mtio(7).

NOTES

rrestore is a link to *restore*.

BUGS

restore can get confused when doing incremental restores from dump tapes that were made on active filesystems.

A level 0 dump must be done after a full restore. Because *restore* runs in user code, it has no control over inode allocation. This results in the files being restored having an inode numbering different from the filesystem that was originally dumped. Thus a full dump must be done to get a new set of directories reflecting the new inode numbering, even though the contents of the files is unchanged, so that later incremental dumps will be correct.

Existing dangling symlinks are modified even if the **e** option is supplied, if the dump tape contains a hard link by the same name.

NAME

rlogin – remote login

SYNOPSIS

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

DESCRIPTION

rlogin connects your terminal on the current local host system to the remote host system *rhost*. The remote username used is the same as your local username, unless you specify a different remote name with the **–l** option or use the *username*@*rhost* format.

The *rlogin* arguments and options are:

rhost — The hostname of the remote system.

username — The user ID to be used on the remote system.

–l *username* — Specifies the user ID to be used on the remote system.

–e*c* — Specifies a different escape character. There is no space separating this option flag and the argument character, *c*.

–L — Allows the *rlogin* session to be run in litout mode. A line of the form **~.** disconnects from the remote host, where **~** is the escape character. A line starting with **~!** starts a shell on the IRIS. Similarly, the line **~^Z** (where ^Z, <Ctrl-z>, is the suspend character) suspends the *rlogin* session if you are using *csh*(1).

–8 — Allows an 8-bit input data path at all times; otherwise parity bits are stripped except when the remote side's stop and start characters are other than ^S/^Q.

Each host has a file */etc/hosts.equiv* that contains a list of remote hosts (equivalent hosts) with which it shares account names. The hostnames must be the standard names as described in *rsh*(1C). When you *rlogin* as the same user on an equivalent host, you do not need to give a password.

Each user can also have a private equivalence list in a file *.rhosts* in his home directory. Each line in this file should contain an *rhost* and a *username* separated by a space, which gives an additional remote host where logins without passwords are permitted. If the originating user is not equivalent to the remote user, the remote host prompts for a login and password as in *login*(1). To avoid some security problems, the *.rhosts* file must be owned by either the remote user or root.

The remote terminal type is the same as your local terminal type (as given in your TERM environment variable). The TERM value **iris-ansi** is converted to **iris-ansi-net** when sent to the host. The terminal or window size is also copied to the remote system if the server supports the option, and changes in size are reflected as well. All echoing takes place at the remote site, so that (except for delays) the *rlogin* is

transparent. Flow control via ^S and ^Q and flushing of input and output on interrupts are handled properly.

SEE ALSO

rsh(1C), hosts(4), rhosts(4).

BUGS

Only the TERM environment variable is propagated. The *rlogin* protocol should be extended to propagate useful variables, such as DISPLAY. (Note that *telnet*(1C) is able to propagate environment variables.)

NAME

rsh – remote shell

SYNOPSIS

/usr/bsd/rsh host [**–l** username] [**–n**] command
/usr/bsd/rsh username@host [**–n**] command

DESCRIPTION

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

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

If you omit *command*, instead of executing a single command, you are logged in on the remote host using *rlogin*(1C). In this case, *rsh* understands the additional arguments to *rlogin*.

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

```
rsh otherhost cat remotefile >> localfile
```

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

```
rsh otherhost cat remotefile ">>" otherremotefile
```

appends *remotefile* to *otherremotefile*.

SEE ALSO

rlogin(1C), hosts(4), rhosts(4).

BUGS

If you use *csh*(1), *rsh* does not work if your *.cshrc* file on the remote host unconditionally executes interactive or output-generating commands. Put these commands inside the following conditional block:

```
if ($?prompt) then

endif
```

so they won't interfere with *rcp*, *rsh*, and other non-interactive, *rcmd*(3N)-based programs.

If you are using *csh*(1) and put a *rsh*(1C) in the background without redirecting its input away from the terminal, it blocks even if no reads are posted by the remote command. If no input is desired, you should use the **−n** option, which redirects the input of *rsh* to */dev/null*.

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

Job control signals stop the local *rsh* process only; this is arguably wrong, but currently hard to fix.

NAME

savecore – save a crash vmcore dump of the operating system

SYNOPSIS

/etc/savecore [**–f**] [**–v**] *dirname* [*system*]

DESCRIPTION

savecore is meant to be called by */etc/rc2.d/S48savecore*. Its functions are to save the core dump of the system (assuming one was made) and to write a reboot message in the shutdown log. The *S48savecore* script specifies *dirname* as */var/adm/crash* by default, unless overridden by site-specific command-line options in the file */etc/config/savecore.options*.

When the system crashes, one of the last steps that the kernel performs is to write the contents of system memory to the dump device. The dump device is */dev/swap*, When the system crashes, the process of creating the dump image will overwrite any data on the dump device. Thus, the dump device must be a raw partition that does not contain any data that needs to be preserved across a system crash (which is why */dev/swap* is the obvious candidate for the dump device).

savecore reads the core image saved on the dump device and saves that core image in the file *dirname*/vmcore.n.

IRIX *savecore* can also write compressed core image files which are named *dirname*/vmcore.n.comp. These compressed files also contain a header which gives certain information about the dump. When copying out a compressed dump, *savecore* also logs the last few messages printed to the console before the system went down. A compressed dump can be expanded with the *uncompvm*(1M) command.

Making sense of any saved core image requires the symbol table of the operating system that was running at the time of the crash. For this reason *savecore* also saves the current default kernel boot file */unix* as *dirname*/unix.n. The trailing ".n" in the pathnames is replaced by a number that 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 bytes on the filesystem that 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*(3C)). If the system crashed as a result of a panic, *savecore* logs the panic string also.

savecore assumes that */unix* corresponds to the running system at the time of the crash. If the core dump was from a system other than */unix*, the name of that system must be supplied as *system*.

The following options apply to *savecore:*

–f Ordinarily, *savecore* checks a magic number on the dump device (usually */dev/swap*) to determine if a core dump was made. This flag forces *savecore* to attempt to save the core image regardless of the state of this magic number. This may be necessary since *savecore* always clears the magic number after reading it. If a previous attempt to save the image failed in some manner, it is still possible to restart the save with this option.

–v Give more verbose output.

DIAGNOSTICS

`warning: /unix may not have created core file`
Printed if *savecore* believes that the system core file does not correspond with the */unix* operating system binary.

`savecore: /unix is not the running system`
Printed for the obvious reason. If the system that crashed was */unix*, use *mv*(1) to change its name before running *savecore*. Use *mv*(1) or *ln*(1) to rename or produce a link to the name of the file of the currently running operating system binary. This enables *savecore* to find name list information about the current state of the running system from the file */unix*.

FILES

/unix	current IRIX
/var/adm/crash	default place to create dump files
/var/adm/crash/bounds	number for next dump file
/var/adm/crash/minfree	minimum filesystem free space
/etc/config/savecore.options	site-specific command-line options

SEE ALSO

uncompvm(1M), nlist(3X).

NAME

setmnt – establish mount table

SYNOPSIS

setmnt [**–f** *mtab*]

DESCRIPTION

setmnt creates the */etc/mtab* table, which is used by the *mount*(1M) and *umount* commands, among others. If given the **–f** option, it creates an alternate *mtab*. *setmnt* reads standard input and writes an entry in *mtab*(4) format for each line read. Input lines have the format:

fsname dir

where *fsname* is the name of the filesystem's *special file* (for example, */dev/dsk/dks?d?s?*) and *dir* is the mountpoint of that filesystem. Thus, *fsname* and *dir* become the first two strings in the mount table entry.

FILES

/etc/mtab

SEE ALSO

devnm(1M), mount(1M).

BUGS

Problems may occur if *fsname* or *dir* is longer than 127 characters.

NAME

sh, jsh, rsh – shell, the standard/job control/restricted command programming language

SYNOPSIS

sh [**–acefhiknprstuvx**] [args]
jsh [**–acefhiknprstuvx**] [args]
/usr/lib/rsh [**–acefhiknprstuvx**] [args]

DESCRIPTION

sh is a command programming language that executes commands read from a terminal or a file.

jsh is an interface to the shell that provides all the functionality of *sh* and enables Job Control (see **Job Control** below).

/usr/lib/rsh is a restricted version of the standard command interpreter *sh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See **Invocation** below for the meaning of arguments to the shell.

Definitions

A *blank* is a tab or a space. A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a name, a digit, or any of the characters *****, **@**, **#**, **?**, **–**, **$**, and **!** **.**

Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec*(2)). The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by **|**. The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **&&**, or **||**, and optionally terminated by **;** or **&**. Of these four symbols, **;** and **&** have equal precedence, which is lower than that of **&&** and **||**. The symbols **&&** and **||** also have equal precedence. A semicolon (**;**) causes sequential execution of the preceding pipeline; an ampersand (**&**) causes asynchronous execution of the preceding pipeline (that is, the shell does *not* wait for that pipeline to finish). The symbol **&&** (**||**) causes the *list* following it to be executed only if the preceding pipeline returns a zero (nonzero) exit status. An arbitrary number of newlines can appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a *simple-command* or one of the following. Unless otherwise stated, the value returned by a command is that of the last *simple-command* executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**
Each time a **for** command is executed, *name* is set to the next *word* taken from the **in** *word* list. If **in** *word* ... is omitted, the **for** command executes the **do** *list* once for each positional parameter that is set (see **Parameter Substitution** below). Execution ends when there are no more words in the list.

case *word* **in** [*pattern* [| *pattern*] ...**)** *list* **;;**] ... **esac**
A **case** command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for filename generation (see **Filename Generation**) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, the **if** command returns a zero exit status.

while *list* **do** *list* **done**
A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, the **while** command returns a zero exit status; **until** can be used in place of **while** to negate the loop termination test.

(*list***)** Execute *list* in a subshell.

{*list***;}** *list* is executed in the current (that is, parent) shell. The **{** must be followed by a space.

name **() {***list***;}**
Define a function that is referenced by *name*. The body of the function is the *list* of commands between **{** and **}**. The *list* can appear on the same line as the **{**. If it does, the **{** and *list* must be separated by a space. The **}** cannot be on the same line as *list*; it must be on a newline. Execution of functions is described below (see **Execution**). The **{** and **}** are unnecessary if the body of the function is a *command* as defined above, under **Commands**.

The following words are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { }

Comments

A word beginning with **#** causes that word and all the following characters up to a newline to be ignored.

Command Substitution

The shell reads commands from the string between two grave accents (**‘’**) and the standard output from these commands can be used as all or part of a word. Trailing newlines from the standard output are removed.

No interpretation is done on the string before the string is read, except to remove backslashes (\) used to escape other characters. Backslashes can be used to escape a grave accent (**‘**) or another backslash (\) and are removed before the command string is read. Escaping grave accents allows nested command substitution. If the command substitution lies within a pair of double quotes (**" ... ‘...‘ ... "**), a backslash used to escape a double quote (**\"**) is removed; otherwise, it is left intact.

If a backslash is used to escape a newline character (**\newline**), both the backslash and the newline are removed (see the later section on "Quoting"). In addition, backslashes used to escape dollar signs (**\$**) are removed. Since no interpretation is done on the command string before it is read, inserting a backslash to escape a dollar sign has no effect. Backslashes that precede characters other than \, **‘**, **"**, **newline**, and **$** are left intact when the command string is read.

Parameter Substitution

The character **$** is used to introduce substitutable *parameters*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Positional parameters can be assigned values by **set**. Keyword parameters (also known as variables) can be assigned values by writing:

name **=** *value* [*name* **=** *value*] ...

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

${***parameter***}** The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is * or @, all the positional parameters, starting with **$1, are substituted (separated by spaces). Parameter **$0** is set from argument zero when the shell is invoked.

${*parameter***:–***word***}** If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*.

${*parameter***:=***word***}** If *parameter* is not set or is null set it to *word*; the value of the parameter is substituted. Positional parameters cannot be assigned to in this way.

${*parameter***:?***word***}** If *parameter* is set and is non-null, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, the message "parameter null or not set" is printed.

${*parameter***:+***word***}** If *parameter* is set and is non-null, substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is null:

```
echo ${d:-'pwd'}
```

If the colon (**:**) is omitted from the above expressions, the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

***** Expands to the positional parameters, beginning with **1**.

@ Expands to the positional parameters beginning with **1**, except when expanded within double quotes, in which case each positional parameter expands as a separate field.

The number of positional parameters in decimal.

– Flags supplied to the shell on invocation or by the **set** command.

? The decimal value returned by the last synchronously executed command.

$** The process number of this shell. **$ reports the process ID of the parent shell in all shell constructs, including pipelines, and in parenthesized subshells.

! The process number of the last background command invoked.

The following parameters are used by the shell:

HOME The default argument (home directory) for the **cd** command, set to the user's login directory by **login**(1) from the password file (see **passwd**(4)).

PATH The search path for commands (see **Execution** below). The user cannot change **PATH** if executing under *rsh*.

CDPATH The search path for the *cd* command.

MAIL If this parameter is set to the name of a mail file *and* the **MAILPATH** parameter is not set, the shell informs the user of the arrival of mail in the specified file.

MAILCHECK
This parameter specifies how often (in seconds) the shell checks for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell checks before each prompt.

MAILPATH
A colon (**:**) separated list of filenames. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each filename can be followed by **%** and a message to be printed when the modification time changes. The default message is "you have mail".

PS1 Primary prompt string, by default ''**$** ''.

PS2 Secondary prompt string, by default ''**>** ''.

IFS Internal field separators, normally **space**, **tab**, and **newline**.

SHACCT If this parameter is set to the name of a file writable by the user, the shell writes an accounting record in the file for each shell procedure executed.

SHELL When the shell is invoked, it scans the environment (see **Environment** below) for this name. If it is found and 'rsh' is the filename part of its value, the shell becomes a restricted shell.

The shell gives default values to **PATH**, **PS1**, **PS2**, **MAILCHECK**, and **IFS**. **HOME** and **MAIL** are set by *login*(1).

Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (those found in **IFS**) and split into distinct arguments where such characters are found. Explicit null arguments (**""** or **''**) are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed. The original whitespace characters (space, tab, and newline) are always considered internal field separators.

Input/Output

A command's input and output can be redirected using a special notation interpreted by the shell. The following can appear anywhere in a *simple-command* or can precede or follow a *command* and are *not* passed on as arguments to the invoked command. Note that parameter and command substitution occurs before *word* or *digit* is used.

<word Use file *word* as standard input (file descriptor 0).

>word Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.

>>word Use file *word* as standard output. If the file exists output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.

<<[**–**]**word**
After parameter and command substitution is done on *word*, the shell input is read up to the first line that literally matches the resulting *word*, or to an end-of-file. If, however, **–** is appended to **<<**:

1. Leading tabs are stripped from *word* before the shell input is read (but after parameter and command substitution is done on *word*).
2. Leading tabs are stripped from the shell input as it is read and before each line is compared with *word*.
3. Shell input is read up to the first line that literally matches the resulting *word*, or to an end-of-file.

If any character of *word* is quoted (see **Quoting**, later), no additional processing is done to the shell input. If no characters of *word* are quoted:

1. Parameter and command substitution occurs.
2. (Escaped) **\newline** is ignored.
3. \ must be used to quote the characters \, **$**, and ‘.

The resulting document becomes the standard input.

<&*digit* Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using **>&***digit*.

<&– The standard input is closed. Similarly for the standard output using **>&–**.

If any of the above is preceded by a digit, the file descriptor that will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

```
... 1>xxx 2>&1
```

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (that is, *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

Using the terminology introduced on the first page, under **Commands**, if a *command* is composed of several *simple commands*, redirection is evaluated for the entire *command* before it is evaluated for each *simple command*. That is, the shell evaluates redirection for the entire *list*, then each *pipeline* within the *list*, then each *command* within each *pipeline*, then each *list* within each *command*.

If a command is followed by **&** the default standard input for the command is the empty file */dev/null*. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Redirection of output is not allowed in the restricted shell.

Filename Generation

Before a command is executed, each command *word* is scanned for the characters *****, **?**, and **[**. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted filenames that match the pattern. If no filename is found that matches the pattern, the word is left unchanged. The character **.** at the start of a filename or immediately following a **/**, as well as the character **/** itself, must be matched explicitly.

***** Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening **[** is a **!**, any character not enclosed is matched.

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

; & ( ) | ^ < > newline space tab

A character can be *quoted* that is, made to stand for itself) by preceding it with a backslash (\) or inserting it between a pair of quote marks (**'** or **"**). During processing, the shell can quote certain characters to prevent them from taking on a special meaning. Backslashes used to quote a single character are removed from the word before the command is executed. The pair **\newline** is removed from a word before command and parameter substitution.

All characters enclosed between a pair of single quote marks (**'**), except a single quote, are quoted by the shell. Backslash has no special meaning inside a pair of single quotes. A single quote can be quoted inside a pair of double quote marks (for example, **"'"**).

Inside a pair of double quote marks (**""**), parameter and command substitution occurs and the shell quotes the results to avoid blank interpretation and filename generation. If **$*** is within a pair of double quotes, the positional parameters are substituted and quoted, separated by quoted spaces (**"$1 $2** ..."); however, if **$@** is within a pair of double quotes, the positional parameters are substituted and quoted, separated by unquoted spaces (**"$1" "$2"** ...). \ quotes the characters \, **'**, **"**, and **$**. The pair **\newline** is removed before parameter and command substitution. If a backslash precedes characters other than \, **'**, **"**, **$**, and newline, then the backslash itself is quoted by the shell.

Prompting

When used interactively, the shell prompts with the value of **PS1** before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt that is, the value of **PS2**) is issued.

Environment

The *environment* (see *environ*(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment (see also **set** −**a**). A parameter can be removed from the environment with the **unset** command. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by **unset**, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any *simple-command* can be augmented by prefixing it with one or more assignments to parameters. Thus these two commands are equivalent (as far as the execution of *cmd* is concerned if *cmd* is not a Special Command):

```
TERM=450 cmd
(export TERM; TERM=450; cmd)
```

If *cmd* is a Special Command, then

```
TERM=45 cmd
```

modifies the **TERM** variable in the current shell.

If the **−k** flag is set, *all* keyword arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and **c**:

```
echo a=b c
set -k
echo a=b c
```

Signals

When a command is run in the background (*cmd* **&**) under **sh**, it can receive INTERRUPT and QUIT signals but ignores them by default. (A background process can override this default behavior via trap or signal. For details, see the description of **trap**, below, or **signal**(2).) When a command is run in the background under **jsh**, however, it does not receive INTERRUPT or QUIT signals.

Otherwise signals have the values inherited by the shell from its parent, with the exception of signal 11 (SIGSEGV). See also the **trap** command below.

Execution

Each time a command is executed, the command substitution, parameter substitution, blank interpretation, input/output redirection, and filename generation listed above are carried out. If the command name matches the name of a defined function, the function is executed in the shell process (note how this differs from the execution of shell procedures). If the command name does not match the name of a defined function, but matches one of the **Special Commands** listed below, it is executed in the shell process. The positional parameters **$1**, **$2**, and so on are set to the arguments of the function. If the command name matches neither a Special Command nor the name of a defined function, a new process is created and an attempt is made to execute the command via *exec*(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (**:**). The default path is:

:/usr/sbin:/usr/bsd:/bin:/usr/bin:/usr/bin/X11

specifying the current directory, */usr/sbin*, */usr/bsd*, */bin*, */usr/bin*, and */usr/bin/X11*, in that order. Note that the current directory is specified by a null pathname. It can appear immediately after the equal sign, between two colon delimiters anywhere in the path list, or at the end of the path list. If the command name contains a **/** the search path is not used; such commands are not executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A subshell is spawned to read it. A parenthesized command is also executed in a subshell.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *execs* later). If the command was found in a relative directory, its location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the **PATH** variable is changed or the **hash −r** command is executed (see below).

Special Commands

Input/output redirection is now permitted for these commands. File descriptor 1 is the default output location. When Job Control is enabled, additional Special Commands are added to the shell's environment (see **Job Control**).

:	No effect; the command does nothing. A zero exit code is returned.
. *file*	Read and execute commands from *file* and return. The search path specified by **PATH** is used to find the directory containing *file*.
break [*n*]	Exit from the enclosing **for** or **while** loop, if any. If *n* is specified break *n* levels.

continue [*n*] Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified resume at the *n*-th enclosing loop.

cd [*arg*] Change the current directory to *arg*. The shell parameter **HOME** is the default *arg*. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (**:**). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a null pathname. It can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a **/** the search path is not used. Otherwise, each directory in the path is searched for *arg*. The *cd* command cannot be executed by *rsh*.

echo [*arg* ...] Echo arguments. See *echo*(1) for usage and description.

eval [*arg* ...] The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...] The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments can appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*] Causes a shell to exit with the exit status specified by *n*. If *n* is omitted, the exit status is that of the last command executed (an end-of-file also causes the shell to exit.)

export [*name* ...] The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, variable names that have been marked for export during the current shell's execution are listed. (Variable names exported from a parent shell are listed only if they have been exported again during the current shell's execution.) Function names are *not* exported.

getopts Use in shell scripts to support command syntax standards (see *intro*(1)); it parses positional parameters and checks for legal options. See *getopts*(1) for usage and description.

hash [**−r**] [*name* ...] For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **−r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *hits* is the number of times a command has been invoked by the shell process. *cost* is a measure of the work required to locate a command in the search path. If a command is found in a "relative" directory in the search path, after changing to that directory, the stored location of that command is recalculated. Commands for which this is done are indicated by an asterisk (*****) adjacent to the *hits* information. *cost* is incremented when the recalculation is done.

limit [**−h**] [*resource* [*maximum-use*]]
Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource*. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given. If the **−h** flag is given, the hard limits are used instead of the current limits. The hard limits impose a ceiling on the values of the current limits. Only the superuser can raise the hard limits, but a user can lower or raise the current limits within the legal range.

Resources controllable currently include *cputime*, the maximum number of cpu-seconds to be used by each process, *filesize*, the largest single file that can be created, *datasize*, the maximum growth of the data region via *sbrk*(2) beyond the end of the program text, *stacksize*, the maximum size of the automatically-extended stack region, *coredumpsize*, the size of the largest core dump created, *memoryuse*, the maximum amount of physical memory a process can have allocated to it at a given time, *descriptors*, the maximum number of open files, and *vmemory*, the maximum total virtual size of the process, including text, data, heap, shared memory, mapped files, stack, and so on.

The *maximum-use* can be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is **k** or **kilobytes** (1024 bytes); a scale factor of **m** or **megabytes** can also be used. For *cputime* the default scaling is **seconds**, while **m** for minutes or **h** for hours, or a time of the form *mm***:***ss* giving minutes and seconds can be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

newgrp [*arg* ...] Equivalent to **exec newgrp** *arg* .... See *newgrp*(1) for usage and description.

pwd Print the current working directory. See *pwd*(1) for usage and description.

read [*name* ...] One line is read from the standard input and, using the internal field separator, **IFS** (normally space or tab), to delimit word boundaries, the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. Lines can be continued using **\newline**. Characters other than **newline** can be quoted by preceding them with a backslash. These backslashes are removed before words are assigned to *names*, and no interpretation is done on the character that follows the backslash. The return code is 0 unless an end-of-file is encountered.

readonly [*name* ...] The given *name*s are marked **readonly** and the values of the these *name*s cannot be changed by subsequent assignment. If no arguments are given, a list of all **readonly** names is printed.

return [*n*] Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

set [−−**aefhkntuvx** [*arg* ...]]

−a Mark variables that are modified or created for export.

−e Exit immediately if a command exits with a nonzero exit status.

−f Disable filename generation.

−h Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).

−k All keyword arguments are placed in the environment for a command, not just those that precede the command name.

−n Read commands but do not execute them.

−t Exit after reading and executing one command.

−u Treat unset variables as an error when substituting.

−v Print shell input lines as they are read.

−x Print commands and their arguments as they are executed.

−− Do not change any of the flags; useful in setting **$1** to −.

Using **+** rather than − causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags can be found in **$−**. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given the values of all names are printed.

shift [*n*] The positional parameters from **$n+1** ... are renamed **$1** .... If *n* is not given, it is assumed to be 1.

test Evaluate conditional expressions. See *test*(1) for usage and description.

times Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ... The command *arg* is to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An error results when an attempt is made to trap signal 11 (SIGSEGV—segmentation fault). If *arg* is absent all trap(s) *n* are reset to their original values. If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *arg* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

type [*name* ...] For each *name*, indicate how it would be interpreted if used as a command name.

ulimit [*n*] Impose a size limit of *n* blocks on files written by the shell and its child processes (files of any size can be read). If *n* is omitted, the current limit is printed. You can lower your own ulimit, but only a superuser (see *su*(1M)) can raise a ulimit.

umask [*nnn*] The user file creation mask is set to *nnn* (see *umask*(1)). If *nnn* is omitted, the current value of the mask is printed.

unlimit [**−h**] [*resource*]
Removes the limitation on *resource*. If no *resource* is specified, then all *resource* limitations are removed. If **−h** is given, the corresponding hard limits are removed. Only the superuser can do this.

unset [*name* ...] For each *name*, remove the corresponding variable or function. The variables **PATH**, **PS1**, **PS2**, **MAILCHECK** and **IFS** cannot be unset.

wait [*n*] Wait for your background process whose process id is *n* and report its termination status. If *n* is omitted, all your shell's currently active background processes are waited for and the return code is zero.

Invocation

If the shell is invoked through *exec*(2) and the first character of argument zero is −, commands are initially read from */etc/profile* and from *$HOME/.profile*, if such files exist. Thereafter, commands are read as described below, which is also the case when the shell is invoked as */bin/sh*. The flags below are interpreted by the shell on invocation only; Note that unless the **−c** or **−s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

−c *string*
If the **−c** flag is present, commands are read from *string*.

−s If the **−s** flag is present or if no arguments remain, commands are read from the standard input. Any remaining arguments specify the positional parameters. Shell output (except for Special Commands) is written to file descriptor 2.

−i If the **−i** flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.

−p If the **−p** flag is present, the shell skips the processing of the system profile (**/etc/profile**) and the user profile (**.profile**) when it starts.

−r If the **−r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

Job Control (jsh)

When the shell is invoked as **jsh**, Job Control is enabled in addition to all of the functionality described previously for **sh**. Typically Job Control is enabled for the interactive shell only. Noninteractive shells typically do not benefit from the added functionality of Job Control.

With Job Control enabled every command or pipeline the user enters at the terminal is called a *job*. All jobs exist in one of the following states: foreground, background, or stopped. These terms are defined as follows: 1) a job in the foreground has read and write access to the controlling terminal; 2) a job in the background is denied read access and has conditional write access to the controlling terminal (see **stty**(1)); 3) a stopped job is a job that has been placed in a suspended state, usually as a result of a **SIGTSTP** signal (see **signal**(2)). Jobs in the foreground can be stopped by INTERRUPT or QUIT signals from the keyboard; background jobs cannot be stopped by these signals.

Every job the shell starts is assigned a positive integer, called a *job number*, which is tracked by the shell and is used, later, as an identifier to indicate a specific job. Additionally the shell keeps track of the *current* and *previous* jobs. The *current job* is the most recent job to be started or restarted. The *previous job* is the first noncurrent job.

The acceptable syntax for a Job Identifier is of the form:

%*jobid*

where *jobid* can be specified in any of the following formats:

% or **+** For the current job.

– For the previous job.

?*string* Specify the job for which the command line uniquely contains *string*.

n For job number *n*, where *n* is a job number.

pref Where *pref* is a unique prefix of the command name (for example, if the command **ls –l foo** were running in the background, it could be referred to as **%ls**); *pref* cannot contain blanks unless it is quoted.

When Job Control is enabled, the following commands are added to the user's environment to manipulate jobs:

bg [**%***jobid* ...] Resumes the execution of a stopped job in the background. If **%***jobid* is omitted the current job is assumed.

fg [**%***jobid* ...] Resumes the execution of a stopped job in the foreground, also moves an executing background job into the foreground. If **%***jobid* is omitted the current job is assumed.

jobs [–**p**|–**l**] [**%***jobid* ...]

jobs –**x** *command* [*arguments*]
Reports all jobs that are stopped or executing in the background. If **%***jobid* is omitted, all jobs that are stopped or running in the background are reported. The following options modify/enhance the output of **jobs**:

–**l** Report the process group ID and working directory of the jobs.

–**p** Report only the process group ID of the jobs.

–**x** Replace any *jobid* found in *command* or *arguments* with the corresponding process group ID, and then execute *command* passing it *arguments*.

kill [–**signal**] **%***jobid*
Builtin version of **kill** to provide the functionality of the **kill** command for processes identified with a *jobid*.

stop **%***jobid* . . . Stops the execution of a background job(s).

suspend Stops the execution of the current shell (but not if it is the login shell).

wait [**%***jobid* ...] **wait** builtin accepts a job identifier. If **%***jobid* is omitted, **wait** behaves as described above under **Special Commands**.

Restricted Shell (/usr/lib/rsh) Only

/usr/lib/rsh is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of **/usr/lib/rsh** are identical to those of **sh**, except that the following are disallowed:

- changing directory (see *cd*(1))
- setting the value of **$PATH**
- specifying path or command names containing **/**
- redirecting output (**>** and **>>**)

The restrictions above are enforced after *.profile* is interpreted.

A restricted shell can be invoked in one of the following ways: (1) *rsh* is the filename part of the last entry in the */etc/passwd* file (see *passwd*(4)); (2) the environment variable **SHELL** exists and *rsh* is the filename part of its value; (3) the shell is invoked and *rsh* is the filename part of argument 0; (4) the shell is invoke with the **–r** option.

When a command to be executed is found to be a shell procedure, */usr/lib/rsh* invokes *sh* to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the *.profile* (see *profile*(4)) has complete control over user actions by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (that is, */usr/rbin*) that can be safely invoked by a restricted shell. IRIX provides a restricted editor, *red*(1).

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a nonzero exit status. If the shell is being used noninteractively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above).

jsh Only

If the shell is invoked as **jsh** and an attempt is made to exit the shell while there are stopped jobs, the shell issues one warning:

```
UX:jsh:WARNING:there are stopped jobs
```

This is the only message. If another exit attempt is made and there are still stopped jobs, they are sent a **SIGHUP** signal from the kernel and the shell is exited.

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

cd(1), echo(1), env(1), getopts(1), intro(1), login(1), newgrp(1), pwd(1), test(1), umask(1), wait(1), dup(2), exec(2), fork(2), getrlimit(2), pipe(2), signal(2), ulimit(2), profile(4).

CAVEATS

Positional parameters have a range of 0 to 9. Attempting to use the positional parameter **$10** gives the contents of **$1** followed by a **0**, which is probably not the desired result.

Words used for filenames in input/output redirection are not interpreted for filename generation (see **Filename Generation**, above). For example, **cat file1 >a*** creates a file with the name **a***.

Because commands in pipelines are run as separate processes, variables set in a pipeline have no effect on the parent shell.

If you get the error message "cannot fork, too many processes", try using the *wait*(1) command to clean up your background processes. If this doesn't help, the system process table is probably full or you have too many active foreground processes. (There is a limit to the number of process ids associated with your login, and to the number the system can keep track of.)

BUGS

Only the last process in a pipeline can be waited for.

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell continues to *exec* the original command. Use the **hash** command to correct this situation.

Prior to IRIX Release 5.0, the **rsh** command invoked the restricted shell. This restricted shell command is **/usr/lib/rsh** and it can be executed by using the full pathname. Beginning with IRIX Release 5.0, the **rsh** command is the remote shell. See **rsh_bsd**(1C).

NAME

statd – network status monitor daemon

SYNOPSIS

/usr/etc/rpc.statd

DESCRIPTION

statd is an intermediate version of the status monitor. It implements a simple protocol that allows applications to monitor the status of other machines. *lockd*(1M) uses *statd* to detect both client and server failures.

statd is started during system initialization if the *chkconfig*(1M) **lockd** flag is set **on**.

Applications use RPC to register machines they want monitored by *statd*. The status monitor maintains a database of machines to track and the corresponding applications to notify of crashes. It also maintains a database of machines to notify upon recovery of its own host machine and a counter of the number of times it has "recovered."

FILES

/var/statmon/sm	machines to monitor
/var/statmon/sm.bak	machines to notify upon recovery
/var/statmon/state	recovery counter (a.k.a. version number)

SEE ALSO

lockd(1M), network(1M), statmon(4).

BUGS

The crash of a site is detected only upon its recovery.

NAME

su – become superuser or another user

SYNOPSIS

su [–] [name [arg ...]]

DESCRIPTION

su allows you to become another user without logging off. The default user *name* is **root** (that is, superuser).

To use *su*, you must supply the appropriate password (except as described below). If the password is correct, *su* executes a new shell with the real and effective user ID set to that of the specified user. The new shell is the program optionally named in the shell field of the specified user's password file entry (see *passwd*(4)), or */bin/sh* if none is specified (see *sh*(1)). To restore normal user ID privileges, type an EOF (<(Ctrl-d>) to the new shell.

su prompts for a password if the specified user's account has one. However, *su* does not prompt you if your user name is **root** or your name is listed in the specified user's *.rhosts* file as:

```
localhost your_name
```

(The hostname of **localhost** is shorthand for the machine's name.)

Any additional arguments given on the command line are passed to the program invoked as the shell. When using programs like *sh*(1), an *arg* of the form **–c** *string* executes *string* via the shell and an arg of **–r** gives the user a restricted shell.

su reads */etc/default/su* to determine default behavior. To change the defaults, the system administrator should edit this file. Recognized values are:

```
SULOG=file   # Use file as the su log file.
CONSOLE=device   # Log successful attempts to su root to device.
SUPATH=path      # Use path as the PATH for root.
PATH=path        # Use path as the PATH for normal users.
SYSLOG=FAIL      # Log to syslog all failures (SYSLOG=FAIL)
                 # or all successes and failures (SYSLOG=ALL).
```

The following statements are true only if the optional program named in the shell field of the specified user's password file entry is like *sh*(1). If the first argument to *su* is a –, the environment is changed to what would be expected if the user actually logged in as the specified user. This is done by invoking the program used as the shell with an *arg0* value whose first character is –, thus causing the system's profile (*/etc/profile*) and then the specified user's profile (*.profile* in the new HOME directory) to be executed.

Otherwise, the environment is passed along with the possible exception of **$PATH**, which is set to

```
/usr/sbin:/usr/bsd:/sbin:/usr/bin:/bin:/etc:/usr/etc:/usr/bin/X11
```

for **root**. Additionally, environment variables of the form of those that are special to *rld*(1) are not passed to the user's program; that is, variable names beginning with either **_RLD** or **LD_LIBRARY**. Note that if the optional program used as the shell is */bin/sh*, the user's *.profile* can check *arg0* for **–sh** or **–su** to determine if it was invoked by *login*(1) or *su*, respectively. If the user's program is other than */bin/sh*, then *.profile* is invoked with an *arg0* of *–program* by both *login* and *su*.

All attempts to become another user using *su* are logged in the log file */var/adm/sulog* by default.

SHARE II ACTIONS

If *su* is invoked with the – option, and the Share II system is installed and enabled, then the new shell executed by *su* is attached to the lnode of the specified user.

If the specified user is not **root**, the lnode attachment may fail due to a non-existent lnode or reaching a memory or process limit, in which case an error message is printed and *su* fails.

EXAMPLES

To become user **bin** while retaining your previously exported environment, execute:

```
su bin
```

To become user **bin** but change the environment to what would be expected if **bin** had originally logged in, execute:

```
su – bin
```

To execute *command* with the temporary environment and permissions of user **bin**, type:

```
su – bin –c "command args"
```

FILES

/etc/passwd	system's password file
/etc/profile	system's initialization script for */bin/sh* users
/etc/cshrc	system's initialization script for */bin/csh* users
$HOME/.profile	*/bin/sh* user's initialization script
$HOME/.cshrc	*/bin/csh* user's initialization script
$HOME/.rhosts	user's list of trusted users
/var/adm/sulog	log file
/etc/default/su	defaults file
/etc/limconf	compiled Share II configuration file (machine readable)

SEE ALSO

env(1), login(1), rld(1), sh(1), cshrc(4), passwd(4), profile(4), rhosts(4), environ(5), share(5).

DIAGNOSTICS

`su: uid` *N*`: cannot attach to lnode` – *reason*.

The lnode attachment failed, so the shell was not executed.

NAME

symmon – kernel symbolic debugger

DESCRIPTION

symmon is a standalone program used to debug the kernel. It is intended to be used only by those involved in writing and debugging device drivers or other parts of the kernel. The implementation of *symmon* is machine dependent and the commands and functionality described here may not apply to all systems.

To use *symmon*, several steps must be taken to prepare the system:

1. *symmon* must be manually installed by the user, because it is not installed on the system as shipped from the factory. This can be done by installing the ''Debugging Kernels'' subsystem in the IDO software development option.

2. Alterations must be done to the file */var/sysgen/system/irix.sm* to build a kernel capable of being debugged; see the comments in that file for details.

3. The program *setsym*(1M) needs to be run on the newly generated kernel to allow *symmon* to recognize symbols in it.

4. *symmon* needs to be installed in the volume header of the root drive with *dvhtool*(1M). This normally happens as part of the software installation process.

symmon is typically used with a terminal as the system console (see *prom*(1M) for information on how to enable a terminal as the console). When a debug kernel is booted, it automatically tries to load *symmon* from the same source. *symmon* can be booted from an alternate device by setting the **dbgname** environment variable in the prom. For example:

```
setenv dbgname scsi(1)disk(1)rdisk(0)partition(8)symmon
```

loads *symmon* from a disk 1, connected to SCSI controller 1.

Once *symmon* is loaded, the system operates normally until *symmon* is triggered by the keyboard or an exceptional condition happens in the kernel that causes it to enter the debugger automatically. To enter *symmon* from the keyboard, type a <Ctrl-a>. *symmon* prompts with **DBG:** and accepts commands described below.

Built-in Commands

symmon has a set of basic commands for setting and clearing breakpoints and examining system state. Not all of the commands listed below are supported on all systems. Some commands take memory addresses as arguments. Addresses can be given directly in decimal, in hex if preceded by **0x**, in binary if preceded with **0b**, as names of functions or data, as names of registers if preceded by **$**, or as a combination of those with **+** and **−**. Some commands take a range of addresses specified as either *ADDR***:***ADDR* for an inclusive range or *ADDR***#***COUNT* for a count of *COUNT* starting at *ADDR*. Commands are listed below:

brk [*ADDR*]
Set a breakpoint at the given address. If no arguments are given, the set of current breakpoints is listed.

bt [*MAX_FRM*]
Print a stack back trace of up to *MAX_FRM* frames. See the discussion about **ubt** below for an alternate form of stack back trace.

c Continue execution from a breakpoint.

cacheflush [*RANGE*]
Flush both the instruction and data caches over the range of address given.

calc
call *ADDR* [*ARGLIST*]
Set up a stack frame and call the procedure at the specified address.

clear Clear the screen.

dis [*RANGE*]
Disassemble instructions in memory over the range specified.

dump [**−b**|**−h**|**−w**] [**−o**|**−d**|**−x**|**−c**] *RANGE*
Dump the contents of memory. The **−b**, **−h**, and **−w** flags can be used to specify byte, halfword, or full word data. The **−o**, **−d**, **−x**, and **−c** flags can be used to specify octal, decimal, hexadecimal, or ASCII data formats.

The specified range of memory to dump can take these forms:

- *base* for a single location
- *base***#***count* for *count* locations starting at *base*
- *base***:***limit* for locations whose addresses are greater than or equal to *base* but less than *limit*

g [**−b**|**−h**|**−w**] [*ADDR*|*$regname*]
Get and display the contents of memory at the address given. If a register name is given, its contents are displayed at the time the kernel was stopped.

goto *ADDR*
Continue execution until the given address or a breakpoint is reached. This is a short hand way to set a breakpoint at an address, continue, and then remove that breakpoint.

help List a short summary of the built-in commands.

hx *NAME*
The symbol table is searched for entries matching *NAME*, and if one is found, its value is printed.

kp [*KPNAME*]
Kernel print command. If no arguments are given, a list of the available kernel print commands is given. If a name is given, that print function is executed. See the discussion on kernel print commands below for more information.

lkaddr *ADDR*
The given address is matched against the symbol table and the symbols near it are listed.

lkup *STRING*
The given string is matched against the symbol table and any symbol with an equal or longer name is printed. This is convenient when you cannot remember the precise symbol name.

msyms *ID*
Print dynamically loaded kernel module's symbols. The module id is found using either the *lboot* **−V** command or the *ml* **list** command. See the *mload*(4) manual page for more information.

nm *ADDR*
The address given is matched against the symbol table and if an exact match is found, the symbolic name is printed. This is a more restrictive version of the **lkaddr** command described above.

p [**−b**|**−h**|**−w**] *ADDR VALUE*
Put the value given into the address given. This causes a write to memory.

printregs
List the contents of the general purpose registers when the kernel was stopped.

quit Restart the PROM.

s [*COUNT*]
Single step the kernel for either one instruction or the given count. If the current instruction is a branch, then both it and the following instruction are executed. The next unexecuted instruction is disassembled when the command completes. After a **step** command is issued, *symmon* enters a command repeat mode where a null command causes another step to be taken. This repeat mode is indicated by a change to the prompt.

S [*COUNT*]
Same as the **step** command above, except that jump-and-link instructions are stepped over.

tlbdump [*RANGE*]
List the contents of the translation lookaside buffer. If specified, the range of TLB entries given is listed. The range should specify a subset of the 64 TLB slots.

tlbflush [*RANGE*]
Flush the TLB over the range of entries given or the entire TLB if no range is specified.

tlbmap [**–i** *INDEX*] [**–n**|**–d**|**–g**|**–v**] *VADDR PADDR*
Inserts an entry in the TLB that maps the virtual address given by *VADDR* to the physical address given by *PADDR*. If specified, the TLB slot given by *INDEX* is used. The **–n**, **–d**, **–g**, and **–v** flags can be used to turn on the non-cached, dirty, global, and valid bits. The current TLB context number is used.

tlbpid [*PID*]
Get or set the current TLB context number. If no argument is given, the current TLB context number is returned; otherwise, the context number is set to the argument.

tlbptov *PADDR*
Display TLB entries that map a virtual address to the physical address given.

tlbvtop *VADDR* [*PID*]
Find the physical address mapped to the virtual address given by *VADDR*. If *PID* is given, then it is used as the TLB context number in the match; otherwise, the current TLB context number is used.

unbrk [*BPNUM*]
Remove the breakpoint with the breakpoint number given. The breakpoint number can by determined by listing the set breakpoints with the **brk** command.

wpt [**r**|**w**|**rw**] [**0**|*PADDR*]
Set a read, write or read/write watch point at on physical address using the R4000 watch point registers. The address must be double word aligned, and the watch point trips on any access within the next eight bytes. An argument of 0 clears the watch point. Note that the R4000 only supports one watch point at a time.

[*ADDR*]/[*COUNT*][**d**|**D**|**o**|**O**|**x**|**X**|**b**|**s**|**c**|**i**]
Dump the contents of memory at the given address. This command functions in a similar manner as the *dbx*(1) command of the same syntax.

Kernel Print Commands

The kernel extends the set of built-in *symmon* commands with kernel print commands. These commands dump various kernel data structures.

proc *PROCINDEX*
Dump the process structure associated with the given process table index. Note that the process table index is not the same as the IRIX process ID.

user *PROCINDEX*
Dump the contents of the user structure for the process with the process table index given.

buf *BUFNUM*
Dump the contents of a buffer structure. The address of the buffer to be dumped is controlled by the **BUFNUM** argument. If **BUFNUM** is a valid K0, K1, or K2 address, then the buffer at that address is displayed. If **BUFNUM** is a small integer, it is used as an index into the buffer table. If **BUFNUM** is equal to –1, summary information about the buffer pool is displayed.

qbuf *DEVICE*
Dump the contents of buffers queued for the device given. The device argument is given as the major/minor device number of the desired device.

pda [*CPUID*]
Dump the contents of the processor private data area for the processor ID given.

runq Dump the run queue. A short summary of each process waiting for CPU time is listed.

eframe [*ADDR*]
The exception frame at the given address is displayed. If the address is a small integer, the exception frame of the process with that process table index is used. The exception frame holds the contents of the general purpose registers at the time the process last executed.

ubt [*PROCINDEX*]
User process stack back trace. A stack back trace is listed for the process whose process table index is given.

plist Process table list. This gives an output similar to *ps*(1) and can be used to find the process table index number for a process.

pb Dump console print buffer. The contents of the console print buffer are printed. This can be useful when an important message has scrolled off the screen.

SEE ALSO

prom(1M).

NAME

systune – display and set tunable parameters

SYNOPSIS

systune [**–bfir**] [**–n** name] [**–p** rootpath]

DESCRIPTION

systune is a tool that enables you to examine and configure your tunable kernel parameters. *systune* can adjust some parameters in real time and informs you if you need to reboot your system after reconfiguration. It saves the reconfigured kernel in */unix.install*, unless the **–f** option is used.

systune has two modes: interactive and noninteractive. Interactive mode allows you to query information about various portions of tunable parameters or to set new values for tunable parameters. Some parameters can be changed while the system is running, and some require a new copy of the kernel to be booted. To enter interactive mode, use the **–i** option. In noninteractive mode, *systune* displays the values of all tunable parameters. Noninteractive mode is the default.

The options are:

–b Both target kernel and the running system are updated with the new values that you specified, if the new values are within the legal range for the parameter specified in */var/sysgen/mtune*. The new values with the corresponding tunable variables are also added into */var/sysgen/stune* file. This is the default behavior.

–f This option forces *systune* to not save the reconfigured kernel in */unix.install*. By default, *systune* tests to see if */unix.install* exists and whether it is identical to the running system. If it is identical, *systune* makes any changes in */unix.install*; otherwise, *systune* copies the current */unix* kernel or the kernel specified by the **–n** option to */unix.install* and makes all changes to the copied kernel. If the copy fails for any reason, such as lack of disk space or the presence of the **–f** option, the currently running kernel is changed.

–i Run *systune* in interactive mode. When *systune* is invoked in interactive mode, no parameter values are immediately displayed. Instead, you see the *systune* prompt:

```
systune->
```

The commands available in interactive mode are described below.

–n *name* This option specifies an alternate kernel *name* to tune in place of **/unix**.

–p *rootpath*
If you specify this option, *rootpath* becomes the starting pathname for *systune* to check for */var/sysgen/stune* and */var/sysgen/mtune*. The default *rootpath* directory is **/**.

–r The new values change on the running system only. If the tunable parameter can not be changed on the running system, nothing is affected. The default is **–b**.

The *systune* commands available in interactive mode are:

quit Quit *systune* immediately. Any changes you have made up to that point are saved and cannot be discarded. You must go through and change back any parameters that you do not wish to be changed.

all Print information on all tunable parameters. This command displays the same information as *systune* invoked in noninteractive mode.

help Show all the built-in commands and group names. *systune* lists two commands (**help** and **all**) and the groups of kernel tunable parameters. Each group of tunable parameters is organized so that related parameters are kept together. For example, the **numproc** parameter group contains parameters related to the number of processes allowed to run on the system at any given time. Its parameters are:

ncsize = 808 (0x328)
ncallout = 40 (0x28)
callout_himark = 332 (0x14c)
ndquot = 808 (0x328)
nproc = 300 (0x12c)

parameter_groupname
Display information for all the tunable parameters in this group along with their values in decimal numerals and in hexadecimal notation.

parameter_name
Display information for this tunable parameter only.

parameter_name newvalue
Set the specified tunable parameter to the new value. For example, to raise the **nproc** parameter in the **numproc** parameter group from 300 to 400, follow this example:

```
systune-> nproc 400
nproc = 300 (0x12c)

Do you really want to change nproc to 400 (0x190)? (y/n) y

In order for the change in parameter nproc to become
effective, /unix.install must be moved to /unix and the
system rebooted.
```

This message tells you that the change does not take effect until a new kernel with the new value is running on your system. *systune* always prints a message to inform you if you need to reboot your system for a kernel change to take effect.

systune makes all requested changes to the kernel in three places, if possible. (Nondynamically adjustable parameters can be changed in only two out of three places.) The parameters are changed in:

- the running kernel image on the workstation
- the */unix* or */unix.install* file
- the */var/sysgen/stune* file

Some sanity checking is performed on the modified kernel parameters to help prevent the creation of kernels that will not function correctly. This checking is performed both by *systune* and by the *lboot*(1M) command. For example, some variables have preset minimum and maximum values. Any attempt to change the variable beyond these threshold values results in an error message, and the variable is not changed.

FILES

/var/sysgen/mtune/*	system tunable parameters
/var/sysgen/stune	local settings for system tunable parameters

SEE ALSO

autoconfig(1M), lboot(1M), mtune(4), stune(4).

NAME

telnet – user interface to the TELNET protocol

SYNOPSIS

telnet [**–d**] [**–n** *tracefile*] [**–l** *user* | **–a**] [**–e** *escape-char*] [*host* [*port*]]

DESCRIPTION

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

Options:

–d Sets the initial value of the **debug** toggle to **TRUE**.

–n *tracefile* Opens *tracefile* for recording trace information. See the **set tracefile** command below.

–l *user* When connecting to the remote system, if the remote system understands the ENVIRON option, **user** is sent to the remote system as the value for the variable USER. This option can also be used with the **open** command.

–a Auto-login. Same as specifying **–l** with your user name. This option can also be used with the **open** command.

–e *escape-char*
Sets the initial *telnet* escape character to *escape-char*. If *escape-char* is the null character (specified by **""** or ''), there is no escape character.

host Indicates the official name, an alias, or the Internet address of a remote host.

port Indicates a port number (address of an application). If a number is not specified, the default **telnet** port is used.

Once a connection has been opened, *telnet* attempts to enable the TELNET LINEMODE option. If this fails, *telnet* reverts to one of two input modes: either "character at a time" or "old line by line" depending on what the remote system supports.

When LINEMODE is enabled, character processing is done on the local system, under the control of the remote system. When input editing or character echoing is to be disabled, the remote system relays that information. The remote system relays changes to any special characters that happen on the remote system, so that they can take effect on the local system.

In "character at a time" mode, most text typed is immediately sent to the remote host for processing.

In "old line by line" mode, all text is echoed locally, and (normally) only completed lines are sent to the remote host. The "local echo character" (initially **^E**) can be used to turn off and on the local echo (this would mostly be used to enter passwords without the password being echoed).

If the LINEMODE option is enabled or if the **localchars** toggle is **TRUE** (the default for "old line by line", see below), the user's **quit**, **intr**, and **flush** characters are trapped locally and sent as TELNET protocol sequences to the remote side. If LINEMODE has ever been enabled, the user's **susp** and **eof** are also sent as TELNET protocol sequences, and **quit** is sent as a TELNET ABORT instead of BREAK. There are options (see **toggle autoflush** and **toggle autosynch** below) that cause this action to flush subsequent output to the terminal (until the remote host acknowledges the TELNET sequence) and flush previous terminal input (in the case of **quit** and **intr**).

While connected to a remote host, *telnet* command mode can be entered by typing the *telnet* "escape character" (initially ^]). When in command mode, the normal terminal editing conventions are available.

The following *telnet* commands are available. Only enough of each command to uniquely identify it needs to be typed (this is also true for arguments to the **mode**, **set**, **toggle**, **unset**, **slc**, **environ**, and **display** commands).

close Close a TELNET session and return to command mode.

display [*argument...*]
Display all, or some, of the **set** and **toggle** values (see below).

mode *type* *type* is one of several options, depending on the state of the TELNET session. The remote host is asked for permission to go into the requested mode. If the remote host is capable of entering that mode, the requested mode is entered.

character Disable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, enter "character at a time" mode.

line Enable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, attempt to enter "old line by line" mode.

isig (–isig)
Attempt to enable (disable) the TRAPSIG mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

edit (–edit)
Attempt to enable (disable) the EDIT mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

softtabs (–softtabs)
Attempt to enable (disable) the SOFT_TAB mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

litecho (–litecho)
Attempt to enable (disable) the LIT_ECHO mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

? Print out help information for the **mode** command.

open *host* [[**–l** *user* | **–a**] [–]*port*]
Open a connection to the named host. If no port number is specified, *telnet* attempts to contact a TELNET server at the default port. The host specification can be either a hostname (see *hosts*(4)) or an Internet address specified in the "dot notation" (see *inet*(3N)). The **–l** option can be used to specify the user name to be passed to the remote system via the ENVIRON option. The **–a** option sends your user name to the remote system via the ENVIRON option. When connecting to a non-standard port, *telnet* omits any automatic initiation of TELNET options. When the port number is preceded by a minus sign, the initial option negotiation is done. After establishing a connection, the *.telnetrc* file in the user's home directory is opened. Lines beginning with a **#** are comment lines. Blank lines are ignored. Lines that begin without whitespace are the start of a machine entry. The first thing on the line is the name of the machine that is being connected to. The rest of the line, and successive lines that begin with whitespace, are assumed to be *telnet* commands and are processed as if they had been typed in manually to the *telnet* command prompt.

quit Close any open TELNET session and exit *telnet*. An end of file (in command mode) also closes a session and exits.

send *arguments*
Send one or more special character sequences to the remote host. The following are the arguments that can be specified (more than one argument can be specified at a time):

abort Send the TELNET ABORT (ABORT processes) sequence.

ao Send the TELNET AO (Abort Output) sequence, which should cause the remote system to flush all output **from** the remote system **to** the user's terminal.

ayt Send the TELNET AYT (Are You There) sequence, to which the remote system may or may not choose to respond.

brk Send the TELNET BRK (Break) sequence, which may have significance to the remote system.

ec Send the TELNET EC (Erase Character) sequence, which should cause the remote system to erase the last character entered.

el Send the TELNET EL (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.

eof Send the TELNET EOF (End Of File) sequence.

eor Send the TELNET EOR (End of Record) sequence.

escape Send the current *telnet* escape character (initially ^**]**).

ga Send the TELNET GA (Go Ahead) sequence, which likely has no significance to the remote system.

getstatus If the remote side supports the TELNET STATUS command, **getstatus** sends the subnegotiation to request that the server send its current option status.

ip Send the TELNET IP (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.

nop Send the TELNET NOP (No OPeration) sequence.

susp Send the TELNET SUSP (SUSPend process) sequence.

synch Send the TELNET SYNCH sequence. This sequence causes the remote system to discard all previously typed (but not yet read) input. This sequence is sent as TCP urgent data (and may not work if the remote system is a 4.2 BSD system — if it doesn't work, a lower case **r** may be echoed on the terminal).

? Print out help information for the **send** command.

set *argument value*

unset *arguments...*

The **set** command sets any one of a number of *telnet* variables to a specific value or to **TRUE**. The special value **off** turns off the function associated with the variable, this is equivalent to using the **unset** command. The **unset** command disables or sets to **FALSE** any of the specified functions. The values of variables can be interrogated with the **display** command. The variables that can be set or unset, but not toggled, are listed here. In addition, any of the variables for the **toggle** command can be explicitly set or unset using the **set** and **unset** commands.

echo The value (initially ^**E**) which, when in "line by line" mode, toggles between doing local echoing of entered characters (for normal processing) and suppressing echoing of entered characters (for entering, say, a password).

eof If *telnet* is operating in LINEMODE or "old line by line" mode, entering this character as the first character on a line causes this character to be sent to the remote system. The initial value of the eof character is taken to be the terminal's **eof** character.

erase If *telnet* is in *localchars* mode (see **toggle localchars** below), and if *telnet* is operating in "character at a time" mode, when this character is typed, a TELNET EC sequence (see **send ec** above) is sent to the remote system. The initial value for the erase character is taken to be the terminal's **erase** character.

escape The *telnet* escape character (initially ^[), which causes entry into *telnet* command mode (when connected to a remote system).

flushoutput
If *telnet* is in *localchars* mode (see **toggle localchars** below) and the **flushoutput** character is typed, a TELNET AO sequence (see **send ao** above) is sent to the remote host. The initial value for the flush character is taken to be the terminal's **flush** character.

interrupt If *telnet* is in *localchars* mode (see **toggle localchars** below) and the **interrupt** character is typed, a TELNET IP sequence (see **send ip** above) is sent to the remote host. The initial value for the interrupt character is taken to be the terminal's **intr** character.

kill If *telnet* is in *localchars* mode (see **toggle localchars** below), **and** if *telnet* is operating in "character at a time" mode, when this character is typed, a TELNET EL sequence (see **send el** above) is sent to the remote system. The initial value for the kill character is taken to be the terminal's **kill** character.

lnext If *telnet* is operating in LINEMODE or "old line by line" mode, this character is taken to be the terminal's **lnext** character. The initial value for the lnext character is taken to be the terminal's **lnext** character.

quit If *telnet* is in *localchars* mode (see **toggle localchars** below) and the **quit** character is typed, a TELNET BRK sequence (see **send brk** above) is sent to the remote host. The initial value for the quit character is taken to be the terminal's **quit** character.

reprint If *telnet* is operating in LINEMODE or "old line by line" mode, this character is taken to be the terminal's **reprint** character. The initial value for the reprint character is taken to be the terminal's **reprint** character.

start If the TELNET TOGGLE-FLOW-CONTROL option has been enabled, this character is taken to be the terminal's **start** character. The initial value for the kill character is taken to be the terminal's **start** character.

stop If the TELNET TOGGLE-FLOW-CONTROL option has been enabled, this character is taken to be the terminal's **stop** character. The initial value for the kill character is taken to be the terminal's **stop** character.

susp If *telnet* is in **localchars** mode, or LINEMODE is enabled, and the **suspend** character is typed, a TELNET SUSP sequence (see **send susp** above) is sent to the remote host. The initial value for the suspend character is taken to be the terminal's **suspend** character.

tracefile The file to which the output, caused by **netdata** or **option** tracing being **TRUE**, is written. If it is set to –, tracing information is written to standard output (the default).

worderase
If *telnet* is operating in LINEMODE or "old line by line" mode, this character is taken to be the terminal's *worderase* character. The initial value for the worderase character is taken to be the terminal's *worderase* character.

slc *state* The **slc** command (Set Local Characters) is used to set or change the state of the special characters when the TELNET LINEMODE option has been enabled. Special characters are characters that get mapped to TELNET commands sequences (like **ip** or **quit**) or line editing characters (like **erase** and **kill**). By default, the local special characters are exported.

export Switch to the local defaults for the special characters. The local default characters are those of the local terminal at the time when *telnet* was started.

import Switch to the remote defaults for the special characters. The remote default characters are those of the remote system at the time when the TELNET connection was established.

check Verify the current settings for the current special characters. The remote side is requested to send all the current special character settings, and if there are any discrepancies with the local side, the local side switches to the remote value.

? Print out help information for the **slc** command.

environ *arguments...*

The **environ** command is used to manipulate the variables that can be sent through the ENVIRON option. The initial set of variables is taken from the user's environment with only the **DISPLAY** and **PRINTER** variables being exported by default.

Valid arguments for the **environ** command are:

define *variable value*

Define the variable *variable* to have a value of *value*. Any variables defined by this command are automatically exported. The *value* can be enclosed in single or double quotes so that tabs and spaces can be included.

undefine *variable*

Remove *variable* from the list of environment variables.

export *variable*

Mark the variable *variable* to be exported to the remote side.

unexport *variable*

Mark the variable *variable* to not be exported unless explicitly asked for by the remote side.

send *variable*

Send the variable *variable* to the remote side.

list List the current set of environment variables. Those marked with a * are sent automatically, other variables are sent only if explicitly requested.

? Print out help information for the **environ** command.

? Display the legal **set** (**unset**) commands.

toggle *arguments...*

Toggle (between **TRUE** and **FALSE**) various flags that control how *telnet* responds to events. These flags can be set explicitly to **TRUE** or **FALSE** using the **set** and **unset** commands listed above. More than one argument can be specified. The state of these flags can be interrogated with the **display** command. Valid arguments are:

autoflush If **autoflush** and **localchars** are both **TRUE**, when the **ao**, **intr**, or **quit** characters are recognized (and transformed into TELNET sequences; see **set** above for details), *telnet* refuses to display any data on the user's terminal until the remote system acknowledges (via a TELNET TIMING MARK option) that it has processed those TELNET sequences. The initial value for this toggle is **TRUE** if the terminal user

had not done an **stty noflsh**, otherwise **FALSE** (see *stty*(1)).

autosynch If **autosynch** and **localchars** are both **TRUE**, when either the **intr** or **quit** characters is typed (see **set** above for descriptions of the **intr** and **quit** characters), the resulting TELNET sequence sent is followed by the TELNET SYNCH sequence. This procedure **should** cause the remote system to begin throwing away all previously typed input until both of the TELNET sequences have been read and acted upon. The initial value of this toggle is **FALSE**.

binary Enable or disable the TELNET BINARY option on both input and output.

inbinary Enable or disable the TELNET BINARY option on input.

outbinary Enable or disable the TELNET BINARY option on output.

crlf If this is **TRUE**, carriage returns are sent as <CR><LF>. If this is **FALSE**, carriage returns are sent as <CR><NUL>. The initial value for this toggle is **FALSE**.

crmod Toggle carriage return mode. When this mode is enabled, most carriage return characters received from the remote host are mapped into a carriage return followed by a linefeed. This mode does not affect those characters typed by the user, only those received from the remote host. This mode is not very useful unless the remote host only sends carriage return, but never linefeed. The initial value for this toggle is **FALSE**.

debug Toggle socket level debugging (useful only to the *superuser*). The initial value for this toggle is **FALSE**.

localchars If this is **TRUE**, the **flush**, **interrupt**, **quit**, **erase**, and **kill** characters (see **set** above) are recognized locally, and transformed into (hopefully) appropriate TELNET control sequences (respectively **ao**, **ip**, **brk**, **ec**, and **el**; see **send** above). The initial value for this toggle is **TRUE** in "old line by line" mode, and **FALSE** in "character at a time" mode. When the LINEMODE option is enabled, the value of **localchars** is ignored and assumed to always be **TRUE**. If LINEMODE has ever been enabled, **quit** is sent as **abort**, and **eof**and **suspend** are sent as **eof**and **susp**, see **send** above).

netdata Toggle the display of all network data (in hexadecimal format). The initial value for this toggle is **FALSE**.

options Toggle the display of some internal *telnet* protocol processing (having to do with TELNET options). The initial value for this toggle is **FALSE**.

prettydump
When the **netdata** toggle is enabled, if **prettydump** is enabled, the output from the **netdata** command is formatted in a more user readable format. Spaces are put between each character in the output, and the beginning of any TELNET escape sequence is preceded by a * to aid in locating them.

? Display the legal **toggle** commands.

z Suspend *telnet*. This command only works when the user is using the *csh*(1).

! [*command*]
Execute a single command in a subshell on the local system. If *command* is omitted, an interactive subshell is invoked.

status Show the current status of **telnet**. This includes the peer one is connected to, as well as the current mode.

? [*command*]
Get help. With no arguments, *telnet* prints a help summary. If a command is specified, *telnet* prints the help information for just that command.

ENVIRONMENT

telnet uses at least the **HOME**, **SHELL**, **USER**, **DISPLAY**, and **TERM** environment variables. Other environment variables can be propagated to the other side via the TELNET ENVIRON option.

FILES

~/.telnetrc user customized telnet startup values

NOTES

On some remote systems, echo has to be turned off manually when in "old line by line" mode.

In "old line by line" mode or LINEMODE the terminal's *eof* character is only recognized (and sent to the remote system) when it is the first character on a line.

NAME

tftpd – internet Trivial File Transfer Protocol server

SYNOPSIS

/usr/etc/tftpd [**–h** *homedir*] [**–l**] [**–n**] [**–s**] [*directory...*]

DESCRIPTION

tftpd is a server that supports the Internet Trivial File Transfer Protocol (TFTP). The TFTP server operates at the port indicated in the **tftp** service description; see *services*(4). The server is normally started by *inetd*(1M).

The use of *tftp*(1C) does not require an account or password on the remote system. Due to the lack of authentication information, *tftpd* allows only publicly readable files to be accessed. Files containing the string **..**/ are not allowed. Files can be written only if they already exist and are publicly writable. Note that this extends the concept of *public* to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling TFTP service. The server should be configured in */etc/inetd.conf* to run as the user ID with the lowest possible privilege.

Relative filenames are looked up in a home directory, */var/boot* by default.

The *tftpd* options are:

–h *homedir*
Changes the home directory to *homedir*, provided it is an absolute pathname.

–l Logs all requests using *syslog*(3C).

–n Suppresses negative acknowledgement of requests for nonexistent or inaccessible relative filenames. Use **–n** when operating on a network with Sun diskless clients that broadcast TFTP requests for bootfiles named by relative pathnames, to avoid storms of negative acknowledgements.

–s Rejects requests to read or write an absolute pathname that does not begin with the home directory prefix and to write a relative pathname. (See below.)

Normally, *tftpd* allows unrestricted access to publicly-readable files in all directories. There are two ways to enhance file security by restricting access to a smaller set of directories. With the **–s** option, *tftpd* rejects requests to read or write an absolute pathname that does not begin with the home directory prefix. It also rejects requests to write a relative pathname. Another method is to restrict access to files in a limited number of *approved* directories by specifying the directory names, *directory*, as arguments to *tftpd* after the other options. For an absolute pathname request, *tftpd* allows the request if its name begins with one of these directories or the home directory. For a relative pathname request, the home directory and the directory list are searched in order. Up to ten directories can be listed if no other command-line options are specified. (*inetd* limits the total number of command-line arguments to ten.)

SEE ALSO

inetd(1M), tftp(1C).

NAME

versions – software versions tool

SYNOPSIS

versions [**–abndpvIV**] [**–r** root] [**display**] [name ...]

versions [**–ckmpstuvxBSUV**] [**–r** root] [listtype] [**user**] [name ...]

versions [**–vFV**] [**–r** root] **remove** name ...

DESCRIPTION

versions calls the programs *showprods*, *showfiles*, and (in the case of removing software with *versions* **remove**), *inst*. Users may wish to use these programs directly, instead of *versions*, since these programs provide a more complete and consistent set of capabilities.

To find out what underlying command *versions* would execute with a given set of arguments, use the **–V** option ahead of other options, for example, **versions -V remove ftn.sw pas.sw**.

The *versions* command has three functions:

– (*showprods*): It displays information about the software that is currently installed on your system and the software that has been available for installation, but is not presently installed. This information is presented at the *product*, *image*, and **subsystem** levels (see the **Definitions** section).

– (*showfiles*): It displays lists of files on your system and information about those files.

– (*inst*): It removes installed software from your system.

The synopsis for each of these three uses of the *versions* command is shown above and the functions are discussed in detail in the sections that follow. In addition, the **Definitions** section defines some terms that are key to understanding and using *versions*. The section called **Updating Configuration Files** explains how to use *versions* to identify and modify files that require site-specific modifications after new software is installed.

Definitions

The *name* argument to *versions* is a *product*, *image*, or *subsystem*. A product is a collection of files that *inst* installs on your system. This collection of files is typically a Silicon Graphics software option such as the Network Filesystem (NFS). Products have short names that are used for installation purposes (for example, **nfs**).

The files in a product are organized into a three-level hierarchy for ease of installation. The highest level is the product level, the next level is the image level, and the third level is the subsystem level. Thus, the files that make up a product such as NFS are grouped into subsystems. The names of these subsystems reflect the hierarchy: *product.image.subsystem*. Some examples for NFS are **nfs.sw.nis** and **nfs.man.relnotes**.

When you enter a *name* argument to *versions*, you can enter a product name (for example, **nfs**), an image name (for example, **nfs.sw**) or a subsystem name (for example, **nfs.sw.nis**), depending on what parts of a product you are interested in. You can also use * as a shell-type wildcard in *name*, but it must be escaped with double quotes (") if you are using *versions* from the shell. For example, if you are at the shell and you want to list information about all of the installed subsystems that have an image name of **man**, you would enter the command:

```
versions "*.man.*"
```

An example of using wildcards from within *inst* is this command to list the **sw** images in **eoe**:

```
versions eoe.sw.*
```

All of the files on a workstation can be divided into two categories: *installed files* and *user files*. The files in a product are called installed files and are put on your system by *inst*. All other files on your system, no matter how they got there, are called user files. There are two types of installed files, system files, and configuration files. System files are modified by the user of the system only in unusual circumstances. Configuration files, on the other hand, are very likely to need modification because they contain information that is often machine-specific or site-specific. On diskless systems, installed files are also *shared* or *unshared* as well as being system files or configuration files.

Because configuration files often contain modifications, *inst* treats them specially during the installation process. If they have not been modified, *inst* removes the old file and installs the new version during software updates. For configuration files that have been modified, one of three things occurs:

- The new version is not installed at all.
- The new version is installed and the old version is renamed by adding the suffix .O (for *older*) to the name.
- The new version is put in a file whose name is created by adding .N (for *newer*) to the original name.

The section, **Updating Configuration Files**, discusses .O and .N files in more detail.

Using versions to List Products, Images, and Subsystems

With no command line options or arguments, *versions* displays one installed software product, image, or subsystem per line for all products currently installed on the system.

- The first column contains an indication of the installation status of the product, image, or subsystem listed on that line. When no command line options are given, the installation status is **I** (installed).
- The second column gives the name of the product, image, or subsystem.

- The third column gives the date of installation.
- The fourth column gives a description of the product, image or subsystem.

The options that follow allow you to change the output:

–I (installed) List currently installed products, images, and subsystems only. (This is the default behavior.)

–a (all) List all the products, images, and subsystems that are installed, that have been installed and then removed, or were available for installation, but not installed. This is also known as "all of the subsystems *inst* has seen since the last time you made filesystems". Products and images that have at least one subsystem installed are marked **I**, otherwise their first column is blank. Installed subsystems are marked **I**. Subsystems that have been installed and later removed are marked **R**. Subsystems that have never been installed are blank in the first column. Older versions of products that have been replaced by a newer version of the product do not appear on any *versions* list.

–n (number) Show the internal version number rather than the date it was installed.

–d (date) Show the creation date rather than the date it was installed.

–v (verbose) Include subsystems in the output. (This is the default.)

–b (brief) Display only products.

–p (pause) Use the built-in pausing mechanism (similar to *more*(1)) after each screenful of output. (This is the default when *versions* is used within *inst*.)

The **–r** option allows you to operate on an IRIX tree rooted at *root*. The default root directory while running under IRIX is **/**, and while in the miniroot is, **/root** (see *inst*(1M)). You might have a different root directory for diskless prototype trees or for test installations that have been done somewhere other than the default of the system's root.

The **display** argument explicitly requests one line per product, image, and subsystem type output from *versions*. It is the default behavior in the absence of a *listtype* argument, the argument **remove**, or one of the options **ckmsuxSU**.

The possible values for the *name* argument are discussed in the **Definitions** section above. If no *name* is given, the default is to display all currently installed software.

Using versions to List Installed Files

The second form of the *versions* command displays lists of filenames. The combination of single character options, *listtype*, the optional **user** argument, and optional *name*s determines the list of files that is displayed. For some values of *listtype*, additional information is also displayed. If you are not superuser, you may not be able to access some files.

The single character options, *listtype* arguments, and other arguments for this form of *versions* are:

–m (modified) List only modified installed files. There are three types of modified installed files: configuration files that the user has changed to be system or site-specific, files that were modified automatically as part of the installation process, and other installed files that the user has changed.

–u (unmodified) List only unmodified installed files.

–B (bad) List only deleted or unreadable installed files.

–c (configuration) List only installed configuration files.

–s List only installed system files.

–t changed
Test for the presence of configuration files that need to be merged or updated. **versions -t changed** invokes **showfiles -c -t -H**, and returns an exit status of zero if any configuration files with an associated .N or .O version were found. Otherwise a non-zero status is returned. See the *showfiles*(1M) reference page for a complete description of this option.

–S (shared) List only diskless client shared files.

–U (unshared) List only unshared files.

–k (checksums) Calculate checksums of user files in the **long** listing.

–p (pause) Use the built-in pausing mechanism (similar to *more*(1)) after each screenful of output. (This is the default when standard output is a terminal.)

–r *root*
(root) Use an IRIX tree rooted at *root*. The default root directory while running under IRIX is **/**, and while in the miniroot is **/root** (see *inst*(1M)). You might have a different root directory for diskless prototype trees or for test installations that have been done somewhere other than the default of the system's root.

list List installed files. This is the default *listtype* if no *listtype* is given, but one of options **ckmusxSU** is given.

long List installed files and include the file type, the checksum (by *sum –r*), the size in blocks at time of installation, the subsystem name, and flags. The file types are:

f Plain file
d Directory
b Block special
c Character special
l Symbolic link

p FIFO, also known as, named pipe

The flags are:

c Configuration file
t Orphaned configuration file (it was installed with a subsystem that has since been removed)
m File is machine-specific (see *inst*(1M) **−m**)

user List user files. This argument can be used by itself, or with the **list** or **long** arguments.

config
List all installed configuration files and any corresponding .N and .O files.

changed
List installed configuration files that have a corresponding .O or .N file and their respective .O or .N files.

name The possible values for the *name* argument are discussed in the **Definitions** section above. If no *name* is given, the default is to display all currently installed files that meet the criteria of the options and arguments.

Using versions to Remove Products, Images, and Subsystems

The *versions* command with the **remove** argument, and one or more *name* arguments, is used to remove most of the files of one or more subsystems. The files that are not removed are modified configuration files.

If removal of the indicated subsystems causes conflicts, *versions* refuses the action, unless the **−F** option is given, in which case no system integrity checking is done, so it is possible to remove subsystems that are critical to the operation of IRIX, the window system or applications that you want to use.

You must be superuser to use **remove**.

Updating Configuration Files

As discussed in the **Definitions** section, some files in a product are called configuration files and are handled specially during installation because they contain system or site-specific information. As a result of this, .O (older) and .N (newer) versions of configuration files may be left on your system after an installation.

When you reboot your system, a check for .O and .N files is done. If any are present, a message is displayed suggesting that you merge configuration files in cases where there are two versions. To do this, first enter the command:

versions changed

If the output contains any .O configuration files:

The .O version of the configuration file is your earlier version. The no-suffix version contains changes that are required for compatibility with the rest of the newly installed software, that increase functionality, or that fix bugs. You should use *diff*(1) or *gdiff*(1) to compare the two versions of the files and transfer information that you recognize as machine or site-specific from the .O version to the no-suffix version.

If you have any .N configuration files:

The .N version of the configuration file is the new version. It contains changes or new features that can be added to the no-suffix version of the configuration file at your option. You should use *diff*(1) or *gdiff*(1) to compare the two versions of the files and add changes that appeared in the new software from the .N version to the no-suffix version if you want them.

After you have examined the .O and .N configuration files and made any changes you want, you can delete the .O and .N versions of the configuration files. If you want to keep them, you should rename them because they might be removed automatically during the next software installation. If you remove all of the .O and .N configuration files, then no message about configuration files appears when you boot your system. The message also stops appearing even if .O or .N files continue to exist after some number of reboots.

NOTES

versions remove fails if there is no space in */usr* to create temporary files.

FILES

/var/inst/hist	binary file containing the file-level installation database of your machine
/var/inst/*product*	binary files containing the product-level installation database of your machine
/var/inst/*	various other files used by *inst*, *swmgr*, *showprods*, and *showfiles*

SEE ALSO

inst(1M), showfiles(1M), showprods(1M), swmgr(1M).

IRIX Admin: Software Installation and Licensing

NAME

xfsrestore – XFS filesystem incremental restore utility

SYNOPSIS

xfsrestore [**–a** housekeeping] [**-e**] [**–f** source ...]
[**–i**] [**–n** file] [**–o**] [**–p** report_interval] [**–r**]
[**–s** subtree ...] [**–t**] [**–v** verbosity] [**–A**] [**–D**]
[**–E**] [**–I** [subopt=value ...]] [**–J**] [**–L** session_label]
[**–O** options_file] [**–Q**] [**–R**] [**–S** session_id] [**–T**]
[**–Y** io_ring_length] [–] destination

DESCRIPTION

xfsrestore restores filesystems from dumps produced by *xfsdump*(1M). Two modes of operation are available: simple and cumulative.

The default is simple mode. *xfsrestore* populates the specified destination directory, *destination*, with the files contained in the dump media.

The **–r** option specifies the cumulative mode. Successive invocations of *xfsrestore* are used to apply a chronologically ordered sequence of delta dumps to a base (level 0) dump. The contents of the filesystem at the time each dump was produced is reproduced. This can involve adding, deleting, renaming, linking, and unlinking files and directories.

A delta dump is defined as either an incremental dump (*xfsdump* **–l** option with level > 0) or a resumed dump (*xfsdump* **–R** option). The deltas must be applied in the order they were produced. Each delta applied must have been produced with the previously applied delta as its base.

–a *housekeeping*
Each invocation of *xfsrestore* creates a directory called *xfsrestorehousekeepingdir*. This directory is normally created directly under the *destination* directory. The **–a** option allows the operator to specify an alternate directory, *housekeeping*, in which *xfsrestore* creates the *xfsrestorehousekeeping* directory. When performing a cumulative (**–r** option) restore, each successive invocation of *xfsrestore* must specify the same alternate directory.

–e Prevents *xfsrestore* from overwriting existing files in the *destination* directory.

–f *source*
Specifies a source of the dump to be restored. This can be the pathname of a device (such as a tape drive), a regular file, or a remote tape drive (see *rmt*(1M)). Up to 20 sources can be specified. All sources are simultaneously applied to the restore. For example, if the dump to be restored spanned three tapes, three tape drives could be used to simultaneously restore the portions of the dump contained on each tape. All other permutations are supported. This option must be omitted if the standard input option (a lone – preceding the *destination* specification) is specified.

–i Selects interactive operation. Once the on-media directory hierarchy has been read, an interactive dialogue is begun. The operator uses a small set of commands to peruse the directory hierarchy, selecting files and subtrees for extraction. The available commands are given below. Initially nothing is selected, except for those subtrees specified with **–s** command line options.

ls [*arg*] List the entries in the current directory or the specified directory, or the specified non-directory file entry. Both the entry's original inode number and name are displayed. Entries that are directories are appended with a '/'. Entries that have been selected for extraction are prepended with a '*'.

cd [*arg*] Change the current working directory to the specified argument, or to the filesystem root directory if no argument is specified.

pwd Print the pathname of the current directory, relative to the filesystem root.

add [*arg*] The current directory or specified file or directory within the current directory is selected for extraction. If a directory is specified, then it and all its descendents are selected. Entries that are selected for extraction are prepended with a '*' when they are listed by **ls**.

delete [*arg*] The current directory or specified file or directory within the current directory is deselected for extraction. If a directory is specified, then it and all its descendents are deselected. The most expedient way to extract most of the files from a directory is to select the directory and then deselect those files that are not needed.

extract Ends the interactive dialogue, and causes all selected subtrees to be restored.

quit *xfsrestore* ends the interactive dialogue and immediately exits, even if there are files or subtrees selected for extraction.

help List a summary of the available commands.

–n *file*
Allows *xfsrestore* to restore only files newer than *file*. The modification time of *file* (i.e., as displayed with the **ls -l** command) is compared to the inode modification time of each file on the source media (i.e., as displayed with the **ls -lc** command). A file is restored from media only if its inode modification time is greater than or equal to the modification time of *file*.

–o Restore file and directory owner/group even if not root. When run with an effective user id of root, *xfsrestore* restores owner and group of each file and directory. When run with any other effective user id it does not, unless this option is specified.

−r Selects the cumulative mode of operation.

−s *subtree*
Specifies a subtree to restore. Any number of **−s** options are allowed. The restore is constrained to the union of all subtrees specified. Each subtree is specified as a pathname relative to the restore *destination*. If a directory is specified, the directory and all files beneath that directory are restored.

−t Displays the contents of the dump, but does not create or modify any files or directories. It may be desirable to set the verbosity level to **silent** when using this option.

−v *verbosity_level*
Specifies the level of detail of the messages displayed during the course of the restore. The argument can be **silent**, **verbose**, or **trace**. The default is **verbose**.

−A Do not restore extended file attributes. If this option is not specified, extended file attributes are restored. Note that dumping of extended file attributes is also optional.

−D Restore DMAPI (Data Management Application Programming Interface) event settings. *xfsdump* backs backs up these settings, but it is usually not desirable to restore them.

−E Prevents *xfsrestore* from overwriting newer versions of files. The inode modification time of the on-media file is compared to the inode modification time of corresponding file in the *destination* directory. The file is restored only if the on-media version is newer than the version in the *destination* directory. The inode modification time of a file can be displayed with the **ls -lc** command.

−I Causes the *xfsdump* inventory to be displayed (no restore is performed). Each time *xfsdump* is used, an online inventory in */var/xfsdump/inventory* is updated. This is used to determine the base for incremental dumps. It is also useful for manually identifying a dump session to be restored (see the **−L** and **−S** options).

−J Inhibits inventory update when on-media session inventory encountered during restore. *xfsrestore* opportunistically updates the online inventory when it encounters an on-media session inventory, but only if run with an effective user id of root and only if this option is not given. Suboptions to filter the inventory display are described later.

−L *session_label*
Specifies the label of the dump session to be restored. The source media is searched for this label. It is any arbitrary string up to 255 characters long. The label of the desired dump session can be copied from the inventory display produced by the **−I** option.

−O *options_file*
Insert the options contained in *options_file* into the beginning of the command line. The options are specified just as they would appear if typed into the command line. In addition, newline characters (\n) can be used as whitespace. The options are placed before all options actually given on the command line, just after the command name. Only one **−O** option can be used. Recursive use is

ignored. The destination directory cannot be specified in *options_file*.

–Q Force completion of an interrupted restore session. This option is required to work around one specific pathological scenario. When restoring a dump session which was interrupted due to an EOM condition and no online session inventory is available, *xfsrestore* cannot know when the restore of that dump session is complete. The operator is forced to interrupt the restore session. In that case, if the operator tries to subsequently apply a resumed dump (using the **–r** option), *xfsrestore* refuses to do so. The operator must tell *xfsrestore* to consider the base restore complete by using this option when applying the resumed dump.

–R Resume a previously interrupted restore. *xfsrestore* can be interrupted at any time by pressing the terminal interrupt character (see *stty*(1)). Use this option to resume the restore. The **–a** and *destination* options must be the same.

–S *session_id*
Specifies the session UUID of the dump session to be restored. The source media is searched for this UUID. The UUID of the desired dump session can be copied from the inventory display produced by the **–I** option.

–T Inhibits interactive dialogue timeouts. *xfsrestore* prompts the operator for media changes. This dialogue normally times out if no response is supplied. This option prevents the timeout.

–X *subtree*
Specifies a subtree to exclude. This is the converse of the **–s** option. Any number of **–X** options are allowed. Each subtree is specified as a pathname relative to the restore *destination*. If a directory is specified, the directory and all files beneath that directory are excluded.

–Y *io_ring_length*
Specify I/O buffer ring length. *xfsrestore* uses a ring of input buffers to achieve maximum throughput when restoring from tape drives. The default ring length is 3.

– A lone – causes the standard input to be read as the source of the dump to be restored. Standard input can be a pipe from another utility (such as *xfsdump*(1M)) or a redirected file. This option cannot be used with the **–f** option. The – must follow all other options, and precede the *destination* specification.

The dumped filesystem is restored into the *destination* directory. There is no default; the *destination* must be specified.

NOTES

Cumulative Restoration

A base (level 0) dump and an ordered set of delta dumps can be sequentially restored, each on top of the previous, to reproduce the contents of the original filesystem at the time the last delta was produced. The operator invokes *xfsrestore* once for each dump. The **–r** option must be specified. The *destination* directory must be the same for all invocations. Each invocation leaves a directory named *xfsrestorehousekeeping* in the

destination directory (however, see the **–a** option above). This directory contains the state information that must be communicated between invocations. The operator must remove this directory after the last delta has been applied.

xfsrestore also generates a directory named *orphanage* in the *destination* directory. *xfsrestore* removes this directory after completing a simple restore. However, if *orphanage* is not empty, it is not removed. This can happen if files present on the dump media are not referenced by any of the restored directories. The *orphanage* has an entry for each such file. The entry name is the file's original inode number, a ".", and the file's generation count modulo 4096 (only the lower 12 bits of the generation count are used).

xfsrestore does not remove the *orphanage* after cumulative restores. Like the *xfsrestorehousekeeping* directory, the operator must remove it after applying all delta dumps.

Media Management

A dump consists of one or more media files contained on one or more media objects. A media file contains all or a portion of the filesystem dump. Large filesystems are broken up into multiple media files to minimize the impact of media dropouts, and to accommodate media object boundaries (end-of-media).

A media object is any storage medium: a tape cartridge, a remote tape device (see *rmt*(1M)), a regular file, or the standard input (currently other removable media drives are not supported). Tape cartridges can contain multiple media files, which are typically separated by (in tape parlance) file marks. If a dump spans multiple media objects, the restore must begin with the media object containing the first media file dumped. The operator is prompted when the next media object is needed.

Media objects can contain more than one dump. The operator can select the desired dump by specifying the dump label (**–L** option), or by specifying the dump UUID (**–S** option). If neither is specified, *xfsrestore* scans the entire media object, prompting the operator as each dump session is encountered.

The inventory display (**–I** option) is useful for identifying the media objects required. It is also useful for identifying a dump session. The session UUID can be copied from the inventory display to the **–S** option argument to unambiguously identify a dump session to be restored.

Dumps placed in regular files or the standard output do not span multiple media objects, nor do they contain multiple dumps.

Inventory

Each dump session updates an inventory database in */var/xfsdump/inventory*. This database can be displayed by invoking *xfsrestore* with the **–I** option. The display uses tabbed indentation to present the inventory hierarchically. The first level is filesystem. The second level is session. The third level is media stream (currently only one stream is supported). The fourth level lists the media files sequentially composing the stream.

Several suboptions are available to filter the display. Specifying **–I depth=***n* (where *n* is 1, 2, or 3) limits the hierarchical depth of the display. Specifying **–I mobjid=***value* (where *value* is a media id) or **–I mobjlabel=***value* (where *value* is a media label) limits the display to media files contained in the specified media object. Similarly, the display can be restricted to a specific filesystem identified by mount point using **–I mnt=***host-qualified_mount_point_pathname*, by filesystem id using **–I fsid=***filesystem_id*, or by device using **–I dev=***host-qualified_device_pathname*.

At most three suboptions can be specified at once: one to constrain the depth, one to constrain the media object, and one to constrain the filesystem. For example, **–I depth=1,mobjlabel="tape 1",mnt=host1:/test_mnt** displays only the filesystem information (depth=1) for those filesystems which were mounted on *host1:/test_mnt* at the time of the dump and only those filesystems dumped to the media object labeled "tape 1".

There is currently no way to remove dumps from the inventory.

An additional media file is placed at the end of each dump stream. This media file contains the inventory information for the current dump session. This is currently unused.

Media Errors

xfsdump is tolerant of media errors, but cannot do error correction. If a media error occurs in the body of a media file, the filesystem file represented at that point is lost. The bad portion of the media is skipped, and the restoration resumes at the next filesystem file after the bad portion of the media.

If a media error occurs in the beginning of the media file, the entire media file is lost. For this reason, large dumps are broken into a number of reasonably sized media files. The restore resumes with the next media file.

FILES

/var/xfsdump/inventory dump inventory database

SEE ALSO

rmt(1M), xfsdump(1M), attr_set(2).

DIAGNOSTICS

The exit code is 0 on normal completion, and non-zero if an error occurred or the restore was terminated by the operator.

BUGS

Pathnames of restored non-directory files (relative to the *destination* directory) must be 1023 characters (MAXPATHLEN) or less. Longer pathnames are discarded and a warning message displayed.

There is no verify option to *xfsrestore*. This would allow the operator to compare a filesystem dump to an existing filesystem, without actually doing a restore.

The interactive commands (**–i** option) do not understand regular expressions.

Cumulative mode (**–r** option) requires that the operator invoke *xfsrestore* for the base and for each delta to be applied in sequence to the base. It would be better to allow the operator to identify the last delta in the sequence of interest, and let *xfsrestore* work backwards from that delta to identify and apply the preceding deltas and base dump, all in one invocation.

NAME

xfs_check – check XFS filesystem consistency

SYNOPSIS

xfs_check [**–i** ino] ... [**–s**] [**–v**] xfs_special

xfs_check **–f** [**–i** ino] ... [**–s**] [**–v**] file

DESCRIPTION

xfs_check checks whether an XFS filesystem is consistent. It is normally run only when there is reason to believe that the filesystem has a consistency problem. The filesystem to be checked is specified by the *xfs_special* argument, which should be the disk or volume device for the filesystem. Filesystems stored in files can also be checked, using the **–f** flag. The filesystem should normally be unmounted or read-only during the execution of *xfs_check*. Otherwise, spurious problems are reported.

The options to *xfs_check* are:

–f Specifies that the special device is actually a file (see the *mkfs_xfs* **–d** *file* option). This might happen if an image copy of a filesystem has been made into an ordinary file.

–s Specifies that only serious errors should be reported. Serious errors are those that make it impossible to find major data structures in the filesystem. This option can be used to cut down the amount of output when there is a serious problem, when the output might make it difficult to see what the real problem is.

–v Specifies verbose output; it is impossibly long for a reasonably-sized filesystem. This option is intended for internal use only.

–i *ino* Specifies verbose behavior for a specific inode. For instance, it can be used to locate all the blocks associated with a given inode.

Any output from *xfs_check* means that the filesystem has an inconsistency. The only repair mechanism available is to dump the filesystem with *xfsdump*(1M), use *mkfs_xfs*(1M) to make a new filesystem, then use *xfsrestore*(1M) to restore the data.

DIAGNOSTICS

Under two circumstances, *xfs_check* unfortunately might dump core rather than produce useful output. First, if the filesystem is completely corrupt, a core dump might be produced instead of the message

xxx `is not a valid filesystem`

Second, if the filesystem is very large (has many files) then *xfs_check* might run out of memory.

The following is a description of the most likely problems and the associated messages. Most of the diagnostics produced are only meaningful with an understanding of the structure of the filesystem.

xxx `is not an XLV volume device name`
The **−d** option is needed for filesystems that reside in disk partitions instead of in XLV volumes.

`agf_freeblks` *n*`, counted` *m* `in ag` *a*
The freeblocks count in the allocation group header for allocation group *a* doesn't match the number of blocks counted free.

`agf_longest` *n*`, counted` *m* `in ag` *a*
The longest free extent in the allocation group header for allocation group *a* doesn't match the longest free extent found in the allocation group.

`agi_count` *n*`, counted` *m* `in ag` *a*
The allocated inode count in the allocation group header for allocation group *a* doesn't match the number of inodes counted in the allocation group.

`agi_freecount` *n*`, counted` *m* `in ag` *a*
The free inode count in the allocation group header for allocation group *a* doesn't match the number of inodes counted free in the allocation group.

`block` *a/b* `expected inum 0 got` *i*
The block number is specified as a pair (allocation group number, block in the allocation group). The block is used multiple times (shared), between multiple inodes. This message usually follows a message of the next type.

`block` *a/b* `expected type unknown got` *y*
The block is used multiple times (shared).

`block` *a/b* `type unknown not expected`
The block is unaccounted for (not in the freelist and not in use).

`link count mismatch for inode` *nnn* `(name` *xxx*`), nlink` *m*`, counted` *n*
The inode has a bad link count (number of references in directories).

`rtblock` *b* `expected inum 0 got` *i*
The block is used multiple times (shared), between multiple inodes. This message usually follows a message of the next type.

`rtblock` *b* `expected type unknown got` *y*
The real-time block is used multiple times (shared).

`rtblock` *b* `type unknown not expected`
The real-time block is unaccounted for (not in the freelist and not in use).

`sb_fdblocks` *n*`, counted` *m*
The number of free data blocks recorded in the superblock doesn't match the number counted free in the filesystem.

`sb_frextents` *n*`, counted` *m*
The number of free real-time extents recorded in the superblock doesn't match the number counted free in the filesystem.

`sb_icount` *n*`, counted` *m*
The number of allocated inodes recorded in the superblock doesn't match the number allocated in the filesystem.

`sb_ifree` *n*`, counted` *m*
The number of free inodes recorded in the superblock doesn't match the number free in the filesystem.

SEE ALSO

mkfs_xfs(1M), xfsdump(1M), xfsrestore(1M), xfs(4).

NAME

xfs_growfs – expand an XFS filesystem

SYNOPSIS

xfs_growfs [**–D** size] [**–d**] [**–e** rtextsize] [**–i**] [**–L** size]
[**–l**] [**–m** maxpct] [**–n**] [**–R** size] [**–r**] [**–x**] mount-point

DESCRIPTION

xfs_growfs expands an existing XFS filesystem (see *xfs*(4)). The *mount-point* argument is the pathname of the directory where the filesystem is mounted. The filesystem must be mounted to be grown (see *mount*(1M)). The existing contents of the filesystem are undisturbed, and the added space becomes available for additional file storage.

The options to *xfs_growfs* are:

–d, **–D** *size*
Specifies that the data section of the filesystem should be grown. If the **–D** *size* option is given, the data section is grown to that size, otherwise the data section is grown to the largest size possible. The size is expressed in filesystem blocks.

–e Allows the real-time extent size to be specified. In *mkfs_xfs*(1M) this is specified with **–r extsize=***nnnn*.

–i The new log is an internal log (inside the data section).

–l, **–L** *size*
Specifies that the log section of the filesystem should be grown, shrunk, or moved. If the **–L** *size* option is given, the log section is changed to be that size, if possible. The size is expressed in filesystem blocks. The size of an internal log must be smaller than the size of an allocation group (this value is printed at *mkfs*(1M) time). If neither **–i** nor **–x** is given with **–l**, the log continues to be internal or external as it was before.

–m Specify a new value for the maximum percentage of space in the filesystem that can be allocated as inodes. In *mkfs_xfs* this is specified with **-i maxpct=***nn*.

–n Specifies that no change to the filesystem is to be made. The filesystem geometry is printed, and argument checking is performed, but no growth occurs.

–r, **–R** *size*
Specifies that the real-time section of the filesystem should be grown. If the **–R** *size* option is given, the real-time section is grown to that size, otherwise the real-time section is grown to the largest size possible. The size is expressed in filesystem blocks. The filesystem does not need to have contained a real-time section before the *xfs_growfs* operation.

–x The new log is an external log (in an XLV log subvolume).

xfs_growfs is most often used in conjunction with logical volumes (see *xlv*(7M) or *lv*(7M)). However, it can also be used on a regular disk partition, for example if a partition has been enlarged while retaining the same starting block.

PRACTICAL USE

Filesystems normally occupy all of the space on the device where they reside. In order to grow a filesystem, it is necessary to provide added space for it to occupy. Therefore there must be at least one spare new disk partition available. Adding the space is done through the mechanism of logical volumes. If the filesystem already resides on a logical volume, the volume is simply extended using *mklv*(1M) or *xlv_mgr*(1M). If the filesystem is currently on a regular partition, it is necessary to create a new logical volume whose first member is the existing partition, with subsequent members being the new partition(s) to be added. Again, *mklv* or *xlv_mgr* is used for this. In either case *xfs_growfs* is run on the mounted filesystem, and the expanded filesystem is then available for use.

SEE ALSO

mkfs_xfs(1M), mklv(1M), mount(1M), xlv_make(1M), lv(7M), xlv(7M).

NAME

xlv_make – create logical volume objects

SYNOPSIS

xlv_make [**−f**] [**−v**] [**−A**] [input_file]

DESCRIPTION

xlv_make creates new logical volume objects by writing logical volume labels to the devices that are to constitute the volume objects. A volume object can be an entire volume, a plex, or a volume element. *xlv_make* allows you to create objects that are not full volumes so that you can maintain a set of spares.

xlv_make supports the following options:

−f Force *xlv_make* to create a *volume element* even if the partition type for the partition specified does not correspond with its intended usage. This is useful, for example, in converting *lv*(7M) volumes to *xlv*(7M) volumes. It is also used to allow creation of objects involving currently mounted partitions.

−v Verbose option. Causes *xlv_make* to generate more detailed output. Also, it causes *xlv_assemble*(1M) to generate output upon exit from *xlv_make*.

−A Do not invoke *xlv_assemble*(1M) upon exit from *xlv_make*. The default is to invoke *xlv_assemble* with the **−q** option unless the **−v** option is specified, in which case *xlv_assemble* is invoked with no options. To invoke other *xlv_assemble* options, specify the **−A** option and invoke *xlv_assemble* manually.

xlv_make only allows you to create volume objects out of disk partitions that are not currently part of other volume objects. Partitions must be of a type suitable for use by *xlv_make*. Suitable types are **xfs**, **efs**, **xlv**, and **xfslog**. Partition types other than these are rejected unless the **−f** command line option or the **ve −force** interactive command is specified. See *fx*(1M) for more information regarding partition types. *xlv_mgr*(1M) must be used to modify or destroy volume objects.

xlv_make can be run either interactively or it can take its commands from an input file, *input_file*. *xlv_make* is written using Tcl. Therefore, all the Tcl features such as variables, control structures, and so on can be used in *xlv_make* commands.

xlv_make creates volume objects by writing the disk labels. To make the newly created logical volumes active, *xlv_assemble*(1M) must be run. *xlv_assemble* is, by default, automatically invoked upon successful exit from *xlv_make*; *xlv_assemble* scans all the disks attached to the system and automatically assembles all the available logical volumes.

Objects are specified top-down and depth-first. You start by specifying the top-level object and continue to specify the pieces that make it up. When you have completed specifying an object at one level, you can back up and specify another object at the same level.

The commands are:

vol *volume_name*
Specifies a volume. The *volume_name* is required. It can be up to 14 characters in length.

log Specifies a log subvolume.

data Specifies a data subvolume.

rt Specifies a real-time subvolume. Real-time subvolumes are used for guaranteed-rate I/O and also for high performance applications that isolate user data on a separate subvolume.

plex [*plex_name*]
Specifies a plex. If this plex is specified outside of a volume, *plex_name* must be given. A plex that exists outside of a volume is known as a standalone plex.

ve [*volume_element_name*] [**−stripe**] [**−concat**] [**−force**]
[**−stripe_unit** *stripe_unit_size*] [**−start** *blkno*] *device_pathnames*
Specifies a volume element. If this volume element is specified outside of a plex, *volume_element_name* must be given.

−stripe Specifies that the data within this volume element is to be striped across all the disks named by *device_pathnames*.

−concat Specifies that all the devices named by *device_pathnames* are to be joined linearly into a single logical range of blocks. This is the default if no flags are specified.

−force Forces the specification of the volume element when the partition type does not agree with the volume element's intended usage. For example, a partition with type **xfslog** could be assigned to a data subvolume. Also, **−force** allows the specification of an object that includes a partition that is currently mounted.

−stripe_unit *stripe_unit_size*
specifies the number of blocks to write to one disk before writing to the next disk in a stripe set. *stripe_unit_size* is expressed in 512-byte blocks. **−stripe_unit** is only meaningful when used in conjunction with **−stripe**. The default stripe unit size, if this flag is not set, is one track. Note: *lv* called this parameter the granularity.

−start *blkno* Specifies that this volume element should start at the given block number within the plex.

end Terminates the specification of the current object.

clear Removes the current, uncompleted object.

show Prints out all the volume objects on the system. This includes existing volume objects (created during an earlier *xlv_make* session) and new objects specified during this session that have not been created (written out to the disk labels) yet.

exit Create the objects specified during this session by writing the disk labels out to all the disks affected, and exit *xlv_make*. In interactive mode, the user is prompted to confirm this action if any new objects have been created.

quit Leave *xlv_make* without creating the specified objects (without writing the disk labels). All the work done during this invocation of *xlv_make* is lost. In interactive mode, the user is prompted to confirm this action if any objects have been specified.

help Displays a summary of *xlv_make* commands.

? Same as **help**.

sh Fork a shell.

EXAMPLES

Example 1

To make a volume from a description in an input file called *volume_config.txt*, give this command:

```
# xlv_make volume_config.txt
```

Example 2

This example shows making some volume objects interactively.

```
# xlv_make
```

Make a spare plex so we can plug it into another volume on demand.

```
xlv_make> plex spare_plex1
spare_plex1
xlv_make> ve /dev/dsk/dks0d2s1 /dev/dsk/dks0d2s2
spare_plex1.0
xlv_make> end
Object specification completed
```

Now make a small volume. (Note that *xlv_make* automatically adds a */dev/dsk* to the disk partition name if it is missing from the *ve* command.)

```
xlv_make> vol small
small
xlv_make> log
small.log
xlv_make> plex
small.log.0
xlv_make> ve dks0d2s3
small.log.0.0
xlv_make> data
small.data
xlv_make> plex
small.data.0
xlv_make> ve dks0d2s14 dks0d2s12
small.data.0.0
xlv_make> end
Object specification completed
xlv_make> show
vol small
ve small.log.0.0        d710aa7d-b21d-1001-868d-080069077725
  start=0, end=1523, (cat)grp_size=1
  /dev/dsk/dks0d2s3 (1524 blks)   d710aa7e-b21d-1001-868d-080069077725
ve small.data.0.0       d710aa81-b21d-1001-868d-080069077725
  start=0, end=4571, (cat)grp_size=2
  /dev/dsk/dks0d2s14 (1524 blks)  d710aa82-b21d-1001-868d-080069077725
  /dev/dsk/dks0d2s12 (3048 blks)  d710aa83-b21d-1001-868d-080069077725
plex spare_plex1
ve spare_plex1.0        d710aa77-b21d-1001-868d-080069077725
  start=0, end=3047, (cat)grp_size=2
  /dev/dsk/dks0d2s1 (1524 blks)   d710aa78-b21d-1001-868d-080069077725
  /dev/dsk/dks0d2s2 (1524 blks)   d710aa79-b21d-1001-868d-080069077725

xlv_make> help
vol volume_name  - Create a volume.
data | log | rt  - Create subvolume of this type.
plex [plex_name] - Create a plex.
ve [-start] [-stripe] [-stripe_unit N] [-force] [volume_element_name] partition(s)
end  - Finished composing current object.
clear- Delete partially created object.
show - Show all objects.
exit - Write labels and terminate session.
quit - Terminate session without writing labels.
help or ? - Display this help message.
sh - Fork a shell.
```

```
xlv_make> exit
#
```

Note that the strings like *d710aa82-b21d-1001-868d-080069077725* shown above are the universally unique identifiers (UUIDs) that identify each XLV object.

Example 3

This example shows a description file that makes the same volume objects as in Example 2.

```
# A spare plex
plex spare_plex1
ve dks0d2s1 dks0d2s2
# A small volume
vol small
log
plex
ve dks0d2s3
data
plex
ve dks0d2s14 dks0d2s12
end
# Write labels before terminating session.
exit
```

Example 4

This example shows making a complex volume interactively. It makes a volume for an XFS filesystem that has a single-partition log and a plexed (mirrored) data subvolume that is striped.

```
# xlv_make
xlv_make> vol movies
movies
xlv_make> log
movies.log
xlv_make> plex
movies.log.0
xlv_make> ve /dev/dsk/dks0d2s1
movies.log.0.0
```

Let the data subvolume have two plexes, each of which consists of two sets of striped disks. The data written to the data subvolume is copied to both movies.data.0 and movies.data.1.

```
xlv_make> data
movies.data
xlv_make> plex
movies.data.0
xlv_make> ve -stripe dks0d1s6 dks0d2s6 dks0d3s6
movies.data.0.0
xlv_make> ve -stripe dks0d4s6 dks0d5s6
movies.data.0.1
xlv_make> plex
movies.data.1
xlv_make> ve -stripe dks1d1s6 dks1d2s6 dks1d3s6
movies.data.1.0
xlv_make> ve -stripe dks1d4s6 dks1d5s6
movies.data.1.1
```

Add a small real-time subvolume. Stripe the data across two disks, with the stripe unit set to 1024 512-byte sectors.

```
xlv_make> rt
movies.rt
xlv_make> plex
movies.rt.0
xlv_make> ve -stripe -stripe_unit 1024 dks4d1s6 dks4d2s6
movies.rt.0.0
xlv_make> end
Object specification completed
xlv_make> exit
#
```

DIAGNOSTICS

`Previous object not completed`
: You have tried to specify a new object before the previous object has been completely specified. For example, the sequence **plex plex** is not valid because the volume elements for the first plex have not been specified yet.

`A volume has not been specified yet`
: This error results from giving **rt**, **data**, or **log** without first specifying a volume to which these subvolumes belong.

`An object with that name has already been specified`
: This error results from giving the **vol** *volume_name*, **plex** *plex_name*, or **ve** *volume_element_name* command when an object with the same name already exists or has been specified in this session.

`A log subvolume has already been specified for this volume`

`A data subvolume has already been specified for this volume`

`A real-time subvolume has already been specified for this volume`
These errors results from giving the **log**, **data**, or **rt** command for a volume that already has a subvolume of the given type.

`A subvolume has not been specified yet`
You have given a **volume** command and then given the **plex** command without first specifying a subvolume to which the plex belongs.

`Too many plexes have been specified for this subvolume`
You have already specified the maximum allowable number of plexes for this subvolume.

`A plex has not been specified yet`
You have given a **ve** command without first giving the **plex** command.

`Too many volume elements have been specified for this plex`
You have reached the maximum number of volume elements that can be in a single plex.

`An error occurred in creating the specified objects`
An error occurred while writing the volume configuration out to the disk labels.

`Unrecognized flag:` *flag*
flag is not recognized.

`Unexpected symbol:` *symbol*
symbol is an unknown command.

`A volume name must be specified`
You have given a **vol** command without giving the name of the volume as an argument.

`Too many disk partitions`
You have specified too many devices for the volume element.

`Cannot determine size of` *partition*`; please verify that the device exists`
xlv_make is unable to figure out the size of the specified disk partition. Make sure that the device exists.

`Unequal partition sizes, truncating the larger partition`
The partitions specified for a striped volume element are not of the same size. This leaves some disk space unusable in the larger partition because data is striped across all the partitions in a volume element.

`A disk partition must be specified`
: You have given the **ve** command without specifying the disk partitions that belong to the volume element as arguments to the command.

`Unknown device: %s`
: You have specified a disk partition that either has no device node in */dev/dsk* or is missing altogether.

`Illegal value`
: The value is out of range for the given flag.

`The volume element's address range must be increasing`
: When you specify the starting offset of a volume element within a plex by using the **ve −start** command, you must specify them in increasing order.

`Disk partition` *partition* `is already being used`
: The disk partition named in the **ve** command is already in use by some other volume object.

`Disk partition` *partition* `is mounted; use ''−force'' to override`
: The disk partition named in the **ve** command is currently mounted. Use of the **−force** argument is required to perform the operation.

`Address range doesn't match corresponding volume element in other plexes`
: A volume element within a plex must have the same address range in all plexes for the subvolume that includes those plexes.

`There are partially specified objects, use ''quit'' to exit without`
: creating them You have entered the **quit** command while there are specified, but not created objects. You should enter **quit** again to really quit at this point and discard specified objects.

`Missing flag value for: %s`
: A command was given that requires an additional argument that was not given.

`Malloc failed`
: There is insufficient memory available for *xlv_make* to operate successfully.

`An error occurred in updating the volume header`
: An attempt to modify a disk's volume header was unsuccessful.

`A striped volume element must have at least two partitions`
: The **ve −stripe** command was given and only one partition was specified.

`Log ve should have partition type xfslog`

`Data ve should have partition type xlv`

`Rt ve should have partition type xlv`

`Standalone object should have partition type xlv or xfslog`

`Mixing partition type xfslog with data types not allowed`
All the partitions that make up a volume element must have the same partition type, either **xlv** or **xfslog**.

`Partition type must be consistent with other ve's in plex`
Partition type does not correspond with intended usage.

`Partition could already belong to lv.`
Check /etc/lvtab A warning that this partition may already belong to an *lv* volume.

`Illegal partition type`
An attempt was made to specify a partition that cannot, under any circumstance, be used in an *xlv*(7M) volume. An example of such a partition would be the volume header.

`Subvolume type does not match any known`
The subvolume being operated on is of no known type.

`Size mismatch`
The partition size information in the volume header does not match that contained in the xlv label.

`Device number mismatch`
A warning that the device number in the xlv label does not match that of the volume header.

`The same partition cannot be listed twice`
The **ve** command was given with the same partition listed twice.

SEE ALSO

xlv_assemble(1M), xlv_labd(1M), xlv_mgr(1M), xlv_plexd(1M), xlv_shutdown(1M), xlvd(1M), xlv(7M).

Tcl and the Tk Toolkit by John K. Ousterhout, Addison-Wesley, 1994.

NOTES

The disk labels created by *xlv_make* are stored only in the volume header of the disks. They do not destroy user data. Therefore, you can make an *lv*(7M) volume into an XLV volume and still preserve all the data on the logical volume.

xlv_make changes the partition type of partitions used in newly created objects to either **xlv** or **xfslog** depending upon their usage.

You must pick a different name for each volume, standalone plex, and standalone volume element. You cannot have, for example, both a volume and a plex named *yy*.

You must be root to run *xlv_make*.

NAME

xlv_mgr – administers XLV logical volume objects and their disk labels

SYNOPSIS

xlv_mgr [**–r** root] [**–v**] [**–x**]

DESCRIPTION

xlv_mgr displays and modifies existing XLV objects (volumes, plexes, volume elements, and XLV disk labels). *xlv_mgr* can operate on XLV volumes even while they are mounted and in use.

xlv_mgr supports the following command line options:

–r *root* Use *root* as the root directory. This is used in the miniroot when / is mounted as */root*.

–v Verbose option. Causes *xlv_mgr* to generate more detailed output.

–x Expert mode. Provides additional functions.

Commands that pertain to plexes are displayed only when the system has been licensed for the plexing portion of XLV.

xlv_mgr provides several types of operations: **attach**, **detach**, **delete**, **change**, **script**, and **show**:

attach Add an XLV object to another XLV object. You can add a volume element to a plex or plex to a volume. The volume element or plex to be added must first be created using *xlv_make*(1M).

detach Separate a part of an XLV object and make it an independent (standalone) XLV object. For example, if you detach a plex from a plexed volume, that plex is separated from the volume and made into a standalone plex. The original volume then has one less plex.

delete Delete an entire XLV object.

script Generate the *xlv_make* commands required to create some or all XLV objects.

show Display the list of XLV objects on the system and their structure.

change Change an attribute associated with can XLV object.

The commands are:

show [**–long**][**–verbose**] **all**

Displays all known XLV objects by name and type. The **–long** option causes more information about each XLV object to be displayed. The **–verbose** displays more detailed information, such as the uuid. The following is an example of the output of this command:

```
xlv_mgr> show all
```

```
Volume:             root_vol (complete)
Volume:             db1 (complete)
Volume Element:     ve12
Plex:               plex2
```

show [**–verbose**] **kernel**
Displays the XLV objects configured into the running kernel. The only XLV objects in the kernel are volumes. Standalone plexes and volume elements are not viable objects in the kernel because they cannot be used. The **–verbose** displays more detailed information on each volume.

show [**–long**] [**–verbose**] **labels** [*device_volume_header*]
Displays XLV disk labels on all disks or the specifed *device_volume_header*. The **–long** option display the secondary label.

show config
Displays XLV software configuration information about the running kernel. For example:

```
xlv_mgr> show config


Allocated subvol locks: 30      locks in use: 8
Plexing license: present
Plexing support: present
Maximum subvol block number: 0x7fffffff
```

show [**–verbose**] **object** *object_name*
Displays detailed information on an XLV object *object_name*. The information includes all the XLV parameters and the disk partitions that make up the object.

In the example below, the volume named *db1* has one subvolume of type **data** that contains two plexes. The first plex has two volume elements, while the second plex only has one volume element. The first volume element in each plex covers the same range of disk blocks. For each volume element, *xlv_mgr* displays the partitions that make up the volume element, the size of the partition, and the range of this volume's disk blocks that map to the volume element. For example:

```
xlv_mgr> show object db1
VOL db1 (complete)
VE db1.data.0.0 [active]
        start=0, end=1100799, (cat)grp_size=1
        /dev/dsk/dks1d4s0 (1100800 blks)
VE db1.data.0.1 [active]
        start=1100800, end=2201599, (cat)grp_size=1
        /dev/dsk/dks1d4s1 (1100800 blks)
VE db1.data.1.0 [active]
```

```
        start=0, end=1100799, (cat)grp_size=1
        /dev/dsk/dks1d4s2 (1100800 blks)
```

attach ve *source dest-plex*
attach ve *source volume*.{**data**|**log**|**rt**}.*N*

The command appends standalone volume element object *source* to the end of destination plex. This enables you to grow a plex or volume by adding a volume element to the end of a plex. You can use this in conjunction with *xfs_growfs*(1M) to grow an XFS filesystem without unmounting it.

Suppose that you have a volume element, *spareve*, that contains a single disk partition */dev/dsk/dks1d4s2*. The following command appends it to plex 0 of the data subvolume of volume *db1*:

```
xlv_mgr> attach ve spareve db1.data.0
xlv_mgr> show object db1
VOL db1 (complete)
VE db1.data.0.0 [active]
        start=0, end=1100799, (cat)grp_size=1
        /dev/dsk/dks1d4s0 (1100800 blks)
VE db1.data.0.1 [active]
        start=1100800, end=2201599, (cat)grp_size=1
        /dev/dsk/dks1d4s1 (1100800 blks)
VE db1.data.0.2 [active]
        start=2201600, end=3302399, (cat)grp_size=1
        /dev/dsk/dks1d4s2 (1100800 blks)
```

attach plex *source volume*.{**data**|**log**|**rt**}

Appends standalone plex object *source* to existing volume *volume*. This command creates duplicate copies of the data on the volume for greater reliability. This operation is sometimes called *mirroring*. After the plex has been added, *xlv_mgr* automatically initiates a plex revive operation; this copies the data from the original XLV plexes to the newly added plex so that the plex holds the same data as the original plexes in the volume. The following appends a plex named *plex2* to the data subvolume of volume *db1*:

```
xlv_mgr> attach plex plex2 db1.data
```

Use the **show object** command to display volume *db1* and see that the disk partitions that were part of *plex2* are now a component of *db2*. *plex2* no longer exists as a standalone plex since it was merged into volume *db1*.

insert ve *source vol*.{**data**|**log**|**rt**}.*N*

insert ve *source plex.N*

Insert standalone volume element object *source* into the destination plex object. This enables you to add a volume element into a gap in a plex.

xlv_mgr requires that the destination be a fully qualified XLV pathname (for example, movies.data.0). The pathname must specify the relative position within the plex to insert the volume element. (The first volume element in a plex is at position 0.) The plex to be operated on can be a standalone plex or a part of a volume. If the plex is part of a volume, the volume, subvolume, and plex must be specified. In the example below it is volume *test*. The following example inserts a volume element *ve5* into a gap in the volume *test*. There is a gap because the first volume element starts at block number 76200. First display the configuration of *test* and *ve5* before inserting *ve5* into *test*.

```
xlv_mgr> show object test

VOL test (has holes)
VE test.data.0.0 [active]
        start=76200, end=152399, (cat)grp_size=1
        /dev/dsk/dks0d2s1 (76200 blks)
VE test.data.0.1 [active]
        start=152400, end=228599, (cat)grp_size=1
        /dev/dsk/dks0d2s2 (76200 blks)

xlv_mgr> show object ve5

VE ve5 [empty]
        start=0, end=76199, (cat)grp_size=1
        /dev/dsk/dks0d2s5 (76200 blks)

xlv_mgr> insert ve ve5 test.data.0

xlv_mgr> show object test

VOL test (complete)
VE test.data.0.0 [stale]
        start=0, end=76199, (cat)grp_size=1
        /dev/dsk/dks0d2s5 (76200 blks)
VE test.data.0.1 [active]
        start=76200, end=152399, (cat)grp_size=1
        /dev/dsk/dks0d2s1 (76200 blks)
VE test.data.0.2 [active]
        start=152400, end=228599, (cat)grp_size=1
        /dev/dsk/dks0d2s2 (76200 blks)
```

detach ve *plex*.*N ve-object*
detach ve *volume*.{**data**|**log**|**rt**}.*N ve-object*
Remove specified volume element from its parent object and save it as *ve-object*. This command separates a volume element from its parent plex. This volume element can later be added to some other XLV object. The plex from which the volume element is removed can be a standalone plex or part of a volume. The detached volume element becomes a standalone XLV volume element object. You must specify the fully qualified pathname of the volume element to be detached and the name to be given to the detached volume element. The detach operation leaves the volume element intact

detach plex *volname*.{**data**|**log**|**rt**}.*plexno plexobject*
This command removes the specified plex from its parent object and save it as *plexobject*. This new standalone plex can later be added back to a volume via the **attach plex** command.

The following example shows how to detach the first plex from a volume:

```
xlv_mgr> detach plex db1.data.0 savedplex
```

delete object *name*
Delete the object *name*. This command enables you to delete a volume, a standalone plex, or a standalone volume element. The XLV configuration is removed from the disks that make up the XLV object. Because the XLV configuration information is stored in the volume header (see *vh*(7M)), this command does not affect any user data that may have been written to the user disk partitions.

delete all[**_labels**]
An expert command, this command deletes the XLV labels from all disks on the local system. You might want to do this to initialize all the disks on a new system and ensure that there is no leftover XLV configuration information on the disks. Note that this is a very dangerous operation. Deleting the disk labels destroys all of the XLV objects on the system.

delete label *device_volume_header*
An expert command, this command deletes the XLV disk label from the named *device_volume_header*.

change nodename *name object ...*
This command changes the nodename associated with the named objects.

The following example shows how to set the node name for the volume *db1* to *homestead*.

```
xlv_mgr> change nodename homestead db1
```

change online *vol*.{**data**|**log**|**rt**}.*N*.*N*
change offline *vol*.{**data**|**log**|**rt**}.*N*.*N*
This command transitions the specified volume element online or offline.

reset Reinitializes *xlv_mgr* data structures by rereading all the XLV configuration information from all the disks.

script [**–write** *filename*] **object** *name*
script [**–write** *filename*] **all**
Generates the required *xlv_make* commands to create the named object or all objects. When the **–write** option is specified, the *xlv_make* commands are saved into *f2filename*.

help Displays a summary of *xlv_mgr* commands.

? Same as **help**.

sh Fork a shell.

quit Terminate this session.

SEE ALSO

xlv_assemble(1M), xlv_make(1M), xlv_plexd(1M), xlv_shutdown(1M), xlv(7M).

Tcl and the Tk Toolkit by John K. Ousterhout, Addison-Wesley, 1994.

NOTES

xlv_mgr operations modify both the XLV disk labels and the kernel data structures as appropriate. This means that you do not need to run *xlv_assemble*(1M) for your changes to take effect. The only exceptions are the XLV label deleting commands **delete all_labels** and **delete label**, which effect only the disk labels.

xlv_mgr automatically initiates plex revive operations (see *xlv_plexd*(1M)) as required after adding a new plex to a volume or a volume element to a plexed volume.

You must be root to run *xlv_mgr*.

NAME

core – format of core image file

SYNOPSIS

#include <core.out.h>

DESCRIPTION

The IRIX system writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called *core* and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID does not produce a core image.

The format of the core image is defined by <*core.out.h*>. It consists of a header, maps, descriptors, and section data.

The header data includes the process name (as in *ps*(1)), the signal that caused the core dump, the descriptor array, and the corefile location of the map array.

Each descriptor defines the length of useful process data. One descriptor defines the general-purpose registers at the time of the core dump for example. The data is present in the core image at the file location given in the descriptor only if the IVALID flag is set in the descriptor.

Each map defines the virtual address and length of a section of the process at the time of the core dump. The data is present in the core image at the file location given in the descriptor only if the VDUMPED flag is set in the map. The process's stack and data sections are normally written in the core image. The process's text is not normally written in the core image.

NOTE

Core image format designed by Silicon Graphics, Inc.

SEE ALSO

dbx(1), ps(1), setuid(2), signal(2).

NAME

efs – layout of the Extent File System

SYNOPSIS

#include <sys/param.h>
#include <sys/fs/efs.h>

DESCRIPTION

An Extent File System can reside on a regular disk partition or on a logical volume; see *lv*(1M). The disk partition or volume is divided into a certain number of 512-byte sectors, also called *basic blocks*. The current maximum size limit of an Extent File System is 16777214 blocks, equivalent to 8 gigabytes.

The Extent File System imposes a common format for certain vital information on its underlying storage medium. Basic block 0 is unused and is available to contain a bootstrap program or other information. Basic block 1 is the *superblock*. The format of an Extent File System superblock is:

```
/*
 * Structure of the superblock for the Extent File System
 */
struct efs {
     /*
      * This portion is read off the volume
      */
     long fs_size;       /* size of filesystem, in sectors */
     long fs_firstcg;    /* bb offset to first cg */
     long fs_cgfsize;    /* size of cylinder group in bb's */
     short fs_cgisize;   /* bb's in inodes per cylinder group */
     short fs_sectors;   /* sectors per track */
     short fs_heads;     /* heads per cylinder */
     short fs_ncg;       /* # of groups in filesystem */
     short fs_dirty;     /* fs needs to be fsck'd */
     time_t fs_time;     /* last superblock update */
     long fs_magic;      /* magic number */
     char fs_fname[6];   /* filesystem name */
     char fs_fpack[6];   /* filesystem pack name */
     long fs_bmsize;     /* size of bitmap in bytes */
     long fs_tfree;      /* total free data blocks */
     long fs_tinode;     /* total free inodes */
     long fs_bmblock;    /* bitmap location */
     long fs_replsb;     /* location of replicated superblock. */
     char fs_spare[24];  /* space for expansion */
     long fs_checksum;   /* checksum of volume portion of fs */
     /*
      * The remainder of this structure, defined fully in
```

```
        * <sys/fs/efs_sb.h> is used by the operating system only.
        */
};
```

The struct efs that is defined in *<sys/fs/efs_sb.h>* contains more fields. The extra fields are used internally by the operating system and are not discussed here. If in doubt, consult the include file for any recent changes to both the section discussed here and changes to relevant definitions.

fs_size holds the size in basic blocks of the filesystem. This variable is filled in when the filesystem is first created with *mkfs*(1M).

fs_firstcg contains the basic block offset to the first *cylinder group*. There are *fs_ncg* cylinder groups contained in the filesystem. Each cylinder group is composed of *fs_cgfsize* basic blocks, of which *fs_cgisize* basic blocks are used for inodes.

fs_sectors and *fs_heads* are used to specify the geometry of the underlying disk containing the filesystem. *fs_heads* is in fact currently unused and should not be relied upon.

fs_dirty is a flag that indicates if the filesystem needs to be checked by the *fsck*(1M) program.

The *fs_time* field contains the time stamp of when the filesystem was last modified.

fs_name holds the *name* of the filesystem (where it is mounted, more or less) and *fs_fpack* contains which volume this filesystem is. The *fs_fpack* field is singularly useless, but is provided for utility compatibility.

fs_magic is used to tag the superblock of the filesystem as an Extent File System. There are two values that are currently used and a macro used to test for either one.

```
#define   EFS_MAGIC 0x072959
#define EFS_NEWMAGIC     0x07295A
#define IS_EFS_MAGIC(x)  ((x == EFS_MAGIC) || (x == EFS_NEWMAGIC))
```

The NEWMAGIC version was added in IRIX 3.3 when the superblock format changed slightly. Filesystems created with that version of *mkfs* or later (or modified with *mkfs* **−r** or extended with *growfs*) get the new magic number; otherwise the older magic number is retained, if present.

The *fs_bmsize* field contains, in bytes, the size of the data block bitmap. The data block bitmap is used for data block allocation. Each one in the bitmap indicates a free block.

The *fs_bmblock* field contains the location of the bitmap if it has been moved from its default location (basic block 2) because the filesystem has been constructed on a logical volume that has been extended (see *growfs*(1M)).

fs_tfree and *fs_tinode* contain the total free blocks and inodes, respectively.

The *fs_replsb* field contains the location of a replicated superblock, if one exists.

The *fs_spare* field is reserved for future use.

Lastly, the *fs_checksum* variable holds a checksum of the above fields (not including itself).

During the *mount*(1M) of the filesystem, the *fs_dirty* and *fs_checksum* fields are examined. If *fs_dirty* is non-zero, or the *fs_checksum* variable does not match the systems computed checksum, the filesystem must be cleaned with *fsck* before it can be mounted. If the filesystem is the *root* partition, this check is ignored, as it is necessary to be able to run *fsck* on a dirty *root* from a dirty *root*. For the format of an inode and its flags, see *inode*(4).

FILES

/usr/include/sys/fs/efs*.h
/usr/include/sys/stat.h

SEE ALSO

fsck(1M), growfs(1M), mkfs(1M), inode(4).

NAME

fstab – static information about filesystems

DESCRIPTION

The file */etc/fstab* describes the filesystems and swapping partitions used by the local machine. The system administrator can modify it with a text editor. It is read by commands that mount, unmount, and check the consistency of filesystems. The file consists of a number of lines of the form:

filesystem directory type options frequency pass

For example:

```
/dev/root   /   efs   rw  0 0
```

Fields are separated by white space; a '#' as the first non-white space character indicates a comment.

The entries from this file are accessed using the routines in *getmntent*(3), which return a structure of the following form:

```
struct mntent {
     char    *mnt_fsname;    /* filesystem name */
     char    *mnt_dir;       /* filesystem path prefix */
     char    *mnt_type;      /* e.g. efs, nfs, proc, or ignore */
     char    *mnt_opts;      /* rw, ro, hard, soft, etc. */
     int     mnt_freq;       /* dump frequency, in days */
     int     mnt_passno;     /* parallel fsck pass number */
};
```

This structure is defined in the *<mntent.h>* include file. To compile and link a program that calls *getmntent*(3), follow the procedures for section (3Y) routines as described in *intro*(3).

The *mnt_dir* field is the full pathname of the directory to be mounted on. The *mnt_type* field determines how the *mnt_fsname* and *mnt_opts* fields are interpreted. Here is a list of the filesystem types currently supported, and the way each of them interprets these fields:

xfs *mnt_fsname* must be a block special device (for example, */dev/root*) or a logical volume.

efs *mnt_fsname* must be a block special device (for example, */dev/root*) or a logical volume.

proc *mnt_fsname* should be the */proc* directory. See *proc*(4).

fd *mnt_fsname* should be the */dev/fd* directory. See *fd*(4).

nfs *mnt_fsname* is the path on the server of the directory to be served. (NFS option only).

cdfs A synonym for type **iso9660** (see below). This type is required for MIPS ABI compliance.

iso9660 *mnt_fsname* must be a generic SCSI device. These are located in the directory */dev/scsi* (for example, */dev/scsi/sc0d7l0*). See *ds*(7M). This filesystem type is used to mount CD-ROM discs in ISO 9660 (with or without Rock Ridge extensions) and High Sierra formats. *eoe2.sw.cdrom* must be installed in order to use the **iso9660** filesystem type.

dos *mnt_fsname* must be a floppy device. These are located in the directory */dev/rdsk* (for example, */dev/rdsk/fds0d2.3.5*). See *smfd*(7M).

hfs *mnt_fsname* must be either a floppy device or a generic SCSI device. Floppy devices are located in the directory */dev/rdsk* (for example, */dev/rdsk/fds0d2.3.5hi*). See *smfd*(7M). SCSI devices are located in the directory */dev/scsi* (for example, */dev/scsi/sc0d4l0*). See *ds*(7M).

swap *mnt_fsname* should be the full pathname to the file or block device to be used as a swap resource.

cachefs *mnt_fsname* should be the filesystem name for the backing filesystem to be mounted as a cache filesystem. This will either be the special filename (for example, */dev/scsi/sc0d7l0*) or *host*:*path*.

rawdata *mnt_fsname* may be the block/char special device of the partition or logical volume to reserve (*mnt_dir* is ignored). This entry enables the system utilities (for example, *mkfs*, *mount*, and so on) to treat the raw partition or logical volume as 'mounted', preventing the partition from inadvertently being overwritten. Any packages that require dedicated raw partitions (databases and so on) should consider placing a **rawdata** entry in *fstab*(4).

If the *mnt_type* is specified as **ignore**, then the entry is ignored. This is useful to show disk partitions not currently used. *mnt_freq* is not used in current IRIX systems.

mnt_passno can be used to control the behavior of parallel filesystem checking on bootup, see *fsck*(1M).

The *mnt_opts* field contains a list of comma-separated option words. Some *mnt_opts* are valid for all filesystem types, while others apply to a specific type only.

Options valid on all filesystems (the default is **rw**) are:

rw Read/write.

ro Read-only.

noauto Ignore this entry during a *mount* **−a** command, to allow the definition of *fstab* entries for commonly-used filesystems that should not be automatically mounted.

grpid Causes a file created within the filesystem to have the group ID of its parent directory, not the creating process's group ID.

nosuid Setuid execution not allowed for non-superusers. This option has no effect for the superuser.

nodev Access to character and block special files is disallowed.

Options specific to **xfs** filesystems are:

dmi Enable the Data Management Interface event callouts.

Options specific to **efs** filesystems (the default is **fsck**, **noquota**) are:

raw=*path* The filesystem's raw device pathname (for example, */dev/rroot*).

fsck *fsck*(1M) invoked with no filesystem arguments should check this filesystem.

nofsck *fsck*(1M) should not check this filesystem by default.

quota Disk quotas enforced.

noquota Disk quotas not enforced.

lbsize=*n* The number of bytes transferred in each read or synchronous write operation.

The value assigned to the **lbsize** option must be a power of two at least as large as the system page size. This value is returned by the *getpagesize*(2) system call and is normally either 4096 or 16384 depending on the system type. The current default for **lbsize** is the largest power of two less than or equal to the size of one disk track. An invalid size will cause the mount to fail with the error EINVAL. Note that less than **lbsize** bytes will be transferred if there are not **lbsize** contiguous bytes of the addressed portion of the file on disk.

Options specific to **iso9660** filesystems (the default is **rw**, which has no effect since CD-ROM discs are always read-only) are:

setx Set execute permission on every file on the mounted filesystem. The default is to make an intelligent guess based on the first few bytes of the file.

notranslate
Don't translate ISO 9660 filenames to UNIX filenames. The default is to convert upper case to lower case and to truncate the part including and after the semicolon.

cache=*blocks*
Set the number of 2048 byte blocks to be used for caching directory contents. The default is to cache 128 blocks.

noext Ignore Rock Ridge extensions. The default when the **noext** option is not specified is to use Rock Ridge extensions if present.

susp Enable processing of System Use Sharing Protocol extensions to the ISO 9660 specification. This is the default.

nosusp Disable processing of System Use Sharing Protocol extensions. This has the same effect as the **noext** option.

rrip Enable processing of the Rock Ridge extensions. This is the default.

norrip Disable processing of the Rock Ridge extensions. This is equivalent to the **noext** option.

nmconv=[clm]
This option is supplied for MIPS ABI compliance; some non-IRIX systems may implement it only for type **cdfs**, IRIX allows it with type **iso9660** also. Only one of the three letters **c**, **l**, or **m** can be specified. This option controls filename translation. **c** has the same meaning as **notranslate** above. **l** requests translation to lower case (the IRIX default), and **m** suppresses the version number (also the IRIX default).

NFS clients can mount **iso9660**, **dos**, and **hfs** filesystems remotely by specifying *hostname*:*mountpoint* for *filesystem* and **nfs** for *type*, where an **iso9660**, **dos**, or **hfs** filesystem is mounted at *mountpoint* on the host *hostname*. In this case, the same *options* apply as with **nfs** (see below).

If the NFS option is installed, the following options are valid for **nfs** filesystems:

vers=*n* Use NFS protocol version *n*. (The default is to try version 3, falling back to version 2 if the version 3 mount fails.)

bg If the first attempt fails, retry in the background.

fg Retry in foreground. (Default)

retry=*n* Set number of mount failure retries to *n*. (Default = 10000)

rsize=*n* Set read buffer size to *n* bytes. (Default = 8K)

wsize=*n* Set write buffer size to *n* bytes. (Default = 8K)

timeo=*n* Set NFS timeout to *n* tenths of a second. (Default = 11)

retrans=*n* Set number of NFS retransmissions to *n*. (Default = 5)

port=*n* Set server UDP port number to *n*. (Default = 2049)

hard Retry request until server responds. (Default)

soft Return error if server doesn't respond.

intr Allow accesses to be interrupted by the following signals: SIGHUP, SIGINT, SIGQUIT, SIGKILL, SIGTERM, and SIGTSTP. (This is ''off'' by default.)

acregmin=*t*
Set the regular file minimum attribute cache timeout to *t* seconds. (Default = 3)

acregmax=*t*
Set the regular file maximum attribute cache timeout to *t* seconds. (Default = 60)

acdirmin=*t*
Set the directory minimum attribute cache timeout to *t* seconds. (Default = 30)

acdirmax=*t*
Set the directory maximum attribute cache timeout to *t* seconds. (Default = 60)

actimeo=*t* Set regular and directory minimum and maximum attribute cache timeouts to *t* seconds.

noac No attribute caching.

private Do not flush delayed writes on last close of an open file, and use local file and record locking instead of a remote lock manager.

shortuid Do not let users with userids or groupids larger than 65535 (see *id*(1M)) create or own files. Some versions of UNIX do not support large userids; trying to create a file with a large userid on such an NFS server can produce undefined and surprising results.

symttl=*t* Set the time-to-live for symbolic links cached by NFS to *t* seconds. **symttl=0** turns off NFS symlink caching. The maximum value for *t* is 3600. (Default = 3600)

asyncnlm Use asynchronous NLM RPC calls. The default is to use synchronous NLM. Using this option requires that *lockd*(1M) be running.

The **bg** option causes *mount* to run in the background if the server's *mountd*(1M) does not respond. *mount* attempts each request **retry=***n* times before giving up.

Once the filesystem is mounted, each NFS request waits **timeo=***n* tenths of a second for a response. If no response arrives, the time-out is multiplied by 2, up to a maximum of MAXTIMO (900), 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.

In the absence of client activity that would invalidate recently acquired file attributes, NFS holds attributes cached for an interval between **acregmin** and **acregmax** for regular files, and between **acdirmin** and **acdirmax** for directories. The **actimeo** option sets all attribute timeout constraints to a given number of seconds. The **noac** option disables attribute caching altogether.

The **private** option greatly improves write performance by caching data and delaying writes on the assumption that only this client modifies files in the remote filesystem. It should be used only if the greater risk of lost delayed-write data in the event of a crash is acceptable given better performance. EFS uses caching strategies similar to private NFS The system reduces the risk of data loss for all filesystems by automatically executing a partial *sync*(2) at regular intervals.

Options specific to **swap** resources are:

pri=*t* Set the priority of the swap device to *t*. The legal values are from 0 to 7 inclusive.

swplo=*t* Set the first 512 byte block to use to *t* (default is 0).

length=*t* Set the number of 512 byte blocks to use to *t* (default is entire file/partition).

maxlength=*t*
Set the maximum number of 512 byte blocks to grow the swap area to *t* (default is to use **length**).

vlength=*t* Set the number of virtual 512 byte blocks to claim this swap file has to *t* (default is to use **length**).

All other options except for *noauto* are ignored for **swap** files.

If the CacheFS option is installed, the following options are valid for **cachefs** filesystems:

backfstype=*file_system_type*
The filesystem type of the back filesystem (for example, **nfs**). Any of the following filesystem types may be used as the back filesystem: *nfs*, *nfs3*, *iso9660*, *dos*, *cdfs*, *kfs*, or *hfs*. If this option is not specified, the back filesystem type is determined from the filesystem name. Filesystem names of the form *hostname*:*path* will be assumed to be type *nfs*.

backpath=*path*
Specifies where the back filesystem is already mounted. If this argument is not supplied, CacheFS determines a mount point for the back filesystem.

cachedir=*directory*
The name of the cache directory.

cacheid=*ID*
ID is a string specifying a particular instance of a cache. If you do not specify a cache ID, CacheFS will construct one.

write-around | non-shared
Write modes for CacheFS. In the **write-around** mode, writes are made to the back filesystem, and the affected file is purged from the cache. Also in this mode, file and record locking is performed through the back filesystem. You can use the **non-shared** mode (the default) when you are sure that no one else will be writing to the cached filesystem. In this mode, all writes are made to both the front and the back filesystem, and the file remains in the cache.

noconst By default, consistency checking is performed. Disable consistency checking by specifying **noconst** only if you mount the filesystem read-only.

private Causes file and record locking to be performed locally. In addition, files remain cached when file and record locking is performed. By default, files are not cached when file and record locking is performed and all file and record locking is handled by the back filesystem.

local-access
Causes the front filesystem to interpret the mode bits used for access checking instead or having the back filesystem verify access permissions.

purge Purge any cached information for the specified filesystem.

suid | nosuid
Allow (default) or disallow set-uid execution.

acregmin=*n*
Specifies that cached attributes are held for at least *n* seconds after file modification. After *n* seconds, CacheFS checks to see if the file modification time on the back filesystem has changed. If it has, all information about the file is purged from the cache and new data is retrieved from the back filesystem. The default value is 30 seconds.

acregmax=*n*
Specifies that cached attributes are held for no more than *n* seconds after file modification. After *n* seconds, all file information is purged from the cache. The default value is 30 seconds.

acdirmin=*n*
Specifies that cached attributes are held for at least *n* seconds after directory update. After *n* seconds, CacheFS checks to see if the directory modification time on the back filesystem has changed. If it has, all information about the directory is purged from the cache and new data is retrieved from the back filesystem. The default value is 30 seconds.

acdirmax=*n*
Specifies that cached attributes are held for no more than *n* seconds after directory update. After *n* seconds, all directory information is purged from the cache. The default value is 30 seconds.

actimeo=*n*
Sets **acregmin**, **acregmax**, **acdirmin**, and **acdirmax** to *n*.

bg This option causes *mount* to run in the background if the back filesystem mount times out.

disconnect
Causes the cache filesystem to operate in disconnected mode when the back filesystem fails to respond. This causes read accesses to files already cached to be fulfilled from the front filesystem even when the back filesystem does not respond.

NOTES

The default *fstab* contains the following entry for the */usr* filesystem:

```
/dev/usr /usr efs rw,noquota,raw=/dev/rusr 0 0
```

The setup program *MAKEDEV* (see *MAKEDEV*(1M)) creates */dev/usr* and */dev/rusr* as links to partition 6 on the root disk. This is the normal disk usage; however, if you wish to set up a machine with the */usr* filesystem residing elsewhere (for example, on a second disk or on a logical volume, described in *lv*(7M)), the *mnt_fsname* field must be changed to the full pathname of the device where the */usr* filesystem actually resides. If present, the path specified by the **raw** option should also be changed to the corresponding full pathname. For example:

```
/dev/dsk/ips0d1s7 /usr efs rw,raw=/dev/rdsk/ips0d1s7 0 0
```

Note that if this is done, the */dev/usr* and */dev/rusr* devices created by *MAKEDEV* do not point to the device containing the */usr* filesystem, and they should not be referenced.

Caution: Do not attempt to reconfigure a system with */usr* in a non-default volume by manually recreating these */dev/usr* and */dev/rusr* links and leaving the *fstab* entry unchanged. While this works in normal operation, it leads to incorrect behavior when installing new software.

The filesystem types **nfs2**, **nfs3**, and **nfs3pref** are accepted for compatibility with earlier releases. **nfs2** is equivalent to **vers=2**. **nfs3** is equivalent to **vers=3**. **nfs3pref** is equivalent to **nfs** with no **vers=** option.

FILES

/etc/fstab

SEE ALSO

cfsadmin(1M), fsck(1M), fsck_cachefs(1M), mount(1M), quotacheck(1M), quotaon(1M), swap(1M), getmntent(3), fd(4), mtab(4), proc(4).

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The */etc/gettydefs* file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login*(1) prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a **<break>** character.

Note: Customers who need to support terminals that pass 8 bits to the system (as is typical outside the U.S.) must modify the entries in */etc/gettydefs* as described in the **WARNINGS** section.

Each entry in */etc/gettydefs* has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, and so on as well as *nnn*, where *nnn* is the octal value of the desired character. The various fields are:

label This is the string against which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).

initial-flags These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. The flags that *getty* understands are the same as the ones listed in */usr/include/sys/termio.h* (see *termio*(7)). Normally only the speed flag is required in the *initial-flags*. *getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*.

final-flags These flags take the same values as the *initial-flags* and are set just before *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab, or newline), they are included in the *login-prompt* field. As a special feature, this field can contain the string **$HOSTNAME**, which is replaced by the current hostname of the machine. See *hostname*(1) for more information.

next-label If this entry does not specify the desired speed, indicated by the user typing a **<break>** character, then *getty* searches for the entry with *next-label* as its *label* field and sets up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set; for instance, **2400** linked to **1200**, which is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of */etc/gettydefs* is used, thus making the first entry of */etc/gettydefs* the default entry. It is also used if *getty* can not find the specified *label*. If */etc/gettydefs* itself is missing, there is one entry built into *getty* that brings up a terminal at **300** baud.

It is strongly recommended that after making or modifying */etc/gettydefs*, it be run through *getty* with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

getty(1M), login(1), stty(1), ioctl(2), termio(7).

WARNINGS

To support terminals that pass 8 bits to the system (see the **BUGS** section), modify the entries in the */etc/gettydefs* file for those terminals as follows: add **CS8** to *initial-flags* and replace all occurrences of **SANE** with the values: **BRKINT IGNPAR ICRNL IXON OPOST ONLCR CS8 ISIG ICANON ECHO ECHOK**.

An example of changing an entry in */etc/gettydefs* is illustrated below. All the information for an entry must be on one line in the file.

Original entry:

```
CONSOLE # B9600 HUPCL OPOST ONLCR # B9600 SANE IXANY TAB3
HUPCL # $HOSTNAME console Login:  # console
```

Modified entry:

```
CONSOLE # B9600 CS8 HUPCL OPOST ONLCR # B9600 BRKINT IGNPAR
ICRNL IXON OPOST ONLCR CS8 ISIG ICANON ECHO ECHOK IXANY
TAB3 HUPCL # $HOSTNAME console Login:  # console
```

This change permits terminals to pass 8 bits to the system so long as the system is in MULTI-USER state. When the system changes to SINGLE-USER state, the *getty* is killed and the terminal attributes are lost. So to permit a terminal to pass 8 bits to the system in SINGLE-USER state, after you are in SINGLE-USER state, type (see *stty*(1)):

```
stty -istrip cs8
```

BUGS

8-bit with parity mode is not supported.

NAME

hosts – hostname-address database

DESCRIPTION

The */etc/hosts* file contains information regarding the known hosts on the network. For each host a single line should be present with the following information:

- Internet address
- official hostname
- aliases (optional)

Items are separated by any number of blanks and/or tab characters. A **#** indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines that search the file. For example,

```
192.0.2.2   iris.widgets.com   iris
```

This file must include entries for all of the machine's network interfaces, the localhost address and a few important machines on the local network. *ifconfig*(1M) uses this file when assigning addresses to the network interfaces during system initialization.

By default, this file is used by *gethostbyname*(3N) and *gethostbyaddr*(3N) only when the NIS or the Berkeley Internet name server (*named*(1M)) are not enabled. The system can be configured to use NIS, *named*, and/or this file, as described in *resolver*(4).

If the host is not connected to any network, the file should contain an entry defining the hostname as an alias for the localhost entry. For example, if the hostname is IRIS, the */etc/hosts* file should contain this line:

```
127.1   localhost   IRIS
```

Sites connected to the Internet should configure the system to use the name server. This file can be created from the official host database maintained at the Network Information Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts. The host database maintained at NIC is incomplete.

Network addresses are specified in the conventional dot (.) notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Legal hostnames can contain any alphanumeric character, the minus sign (–) and period (.). Periods are not part of the name but serve to separate components of a domain-style name.

FILES

/etc/hosts

SEE ALSO

ifconfig(1M), named(1M), gethostbyname(3N), resolver(4), sys_id(4), hostname(5).

NAME

inittab – script for the init process

DESCRIPTION

The */etc/inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process */etc/getty* that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

id:rstate:action:process

Each entry is started with a character other than **#** and ended by a newline. Lines starting with **#** are ignored. A backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments can be inserted in the *process* field using the *sh*(1) convention for comments. Comments in the *process* field of lines that spawn *getty*s are displayed by the *who*(1) command. Such *process* field comments can contain information about the line such as its location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This field, of up to four characters, is used to uniquely identify an entry.

rstate This defines the run-level in which this entry is to be processed. Run-levels effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a run-level or run-levels in which it is allowed to exist.

The run-levels are represented by the letter **s** (or **S**), or a number ranging from **0** through **6**. As an example, if the system is in run-level **1**, only those entries having a **1** in the *rstate* field are processed.

When *init* is requested to change run-levels, all processes that do not have an entry in the *rstate* field for the target run-level are sent the warning signal (**SIGTERM**) and allowed a grace period (see *init*(1M) for the length of this grace period), before being forcibly terminated by a kill signal (**SIGKILL**).

The *rstate* field can define multiple run-levels for a process by selecting more than one run-level in any combination from **0–6**, **s**, and **S**. If no run-level is specified, the process is assumed to be valid at all run-levels.

There are three other values, **a**, **b**, and **c**, that can appear in the *rstate* field, even though they are not true run-levels. Entries that have these characters in the *rstate* field are processed only when the *telinit* (see *init*(1M)) process requests them to be run (regardless of the current run-level of the system). They differ from run-levels in that *init* can never enter run-level **a**, **b**, or **c**. Also, a request for the execution of any of these processes does not change the current run-level. Furthermore, a process started by an **a**, **b**, or **c** command is not killed when *init* changes levels.

They are only killed if their line in */etc/inittab* is marked **off** in the *action* field, their line is deleted entirely from */etc/inittab*, or *init* goes into the SINGLE USER state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The *action*s recognized by *init* are as follows:

respawn If the process does not exist then start the process. Do not wait for its termination (continue scanning the *inittab* file) and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the run-level that matches the entry's *rstate*, start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same run-level causes *init* to ignore this entry.

once Upon *init*'s entering a run-level that matches the entry's *rstate*, start the process, do not wait for its termination. When it dies, do not restart the process. If upon entering a new run-level, the process is still running from a previous run-level change, the program is not restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *init* is to start the process and not wait for its termination. When it dies, *init* does not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s run-level at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed the first time *init* goes from single-user to multi-user state after the system is booted. (If **initdefault** is set to **2**, the process runs right after the boot.) *init* starts the process, waits for its termination and, when it dies, does not restart the process.

powerfail Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR,** see *signal*(2)).

powerwait Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with run-levels. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault An entry with this *action* is only scanned when *init* initially invoked. *init* uses this entry, if it exists, to determine which run-level to enter initially. It does this by taking the highest run-level specified in the *rstate* field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* enters run-level **6**. Additionally, if *init* does not find an **initdefault** entry in */etc/inittab*, it requests an initial run-level from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console (before the **Console Login:** prompt). It is expected that this entry will be used only to initialize devices on which *init* might try to ask the run-level question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh −c 'exec** *command***'**. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the **; #***comment* syntax.

FILES

/etc/inittab

SEE ALSO

getty(1M), init(1M), sh(1), who(1), exec(2), open(2), signal(2).

NAME

inode – format of an Extent File System inode

SYNOPSIS

#include <sys/param.h>
#include <sys/fs/efs_ino.h>

DESCRIPTION

An *inode* is the volume data structure used by the Extent File System (EFS) to implement the abstraction of a file. (This is not to be confused with the *in-core inode* used by the operating system to manage memory-resident EFS files.)

An inode contains the type (for example, plain file, directory, symbolic link, or device file) of the file; its owner, group, and public access permissions; the owner and group ID numbers; its size in bytes; the number of links (directory references) to the file; and the times of last access and last modification to the file. In addition, there is a list of data blocks claimed by the file.

An inode under the Extent File System has the following structure.

```
#define   EFS_DIRECTEXTENTS   12

/*
 * Extent based filesystem inode as it appears on disk.
 * The efs inode is 128 bytes long.
 */
struct    efs_dinode {
     ushort    di_mode;       /* type and access permissions */
     short     di_nlink;      /* number of links */
     ushort    di_uid;        /* owner's user ID number */
     ushort    di_gid;        /* group's group ID number */
     off_t     di_size;       /* number of bytes in file */
     time_t    di_atime;      /* time of last access (to contents) */
     time_t    di_mtime;      /* of last modification (of contents) */
     time_t    di_ctime;      /* of last modification to inode */
     long      di_gen;        /* generation number */
     short     di_numextents; /* # of extents */
     u_char    di_version;    /* version of inode */
     u_char    di_spare;      /* UNUSED */
     union {
          extent    di_extents[EFS_DIRECTEXTENTS];
          dev_t     di_dev;   /* device for IFCHR/IFBLK */
     } di_u;
};
```

The types *ushort*, *off_t*, *time_t*, and *dev_t* are defined in *types*(5). The *extent* type is defined as follows:

```
typedef struct extent {
     unsigned int
          ex_magic:8,    /* magic #, must be 0 */
          ex_bn:24,      /* bb # on volume */
          ex_length:8,   /* length of this extent in bb's */
          ex_offset:24;  /* logical file offset in bb's */
} extent;
```

di_mode contains the type of the file (plain file, directory, and so on), and its read, write, and execute permissions for the file's owner, group, and public. *di_nlink* contains the number of links to the inode. Correctly formed directories have a minimum of two links: a link in the directory's parent and the '.' link in the directory itself. Additional links may be caused by '..' links from subdirectories.

di_uid and *di_gid* contain the user ID and group ID of the file (used to determine which set of access permissions apply: owner, group, or public). *di_size* contains the length of the file in bytes.

di_atime is the time of last access to the file's contents. *di_mtime* is the time of last modification of the file's contents. *di_ctime* is the time of last modification of the inode, as opposed to the contents of the file it represents. These times are given in seconds since the beginning of 1970 GMT.

di_gen is the inode generation number used to sequence instantiations of the inode.

An extent descriptor maps a logical segment of a file to a physical segment (extent) on the volume. The physical segment is characterized by a starting address and a length, both in basic blocks (of 512 bytes) and a logical file offset, also in basic blocks.

di_numextents is the number of extents claimed by the file. If it is less than or equal to *EFS_DIRECTEXTENTS* then the extent descriptors appear directly in the inode as *di_u.di_extents[0 .. di_numextents-1]*. When the number of extents exceeds this range, then *di_u.di_extents[0 .. di_u.di_extents[0].ex_offset-1]* are indirect extents that map blocks holding extent information. There are at most *EFS_DIRECTEXTENTS* indirect extents.

If the inode is a block or character special inode, *di_u.di_numexents* is 0, and *di_u.di_dev* contains a number identifying the device.

If the inode is a symbolic link and *di_u.di_numexents* is 0, the symbolic link path string is stored in the extent descriptor area of the inode. A symbolic link is created with in-line data only when the data string fits within the extent descriptor area, and the tuneable parameter **efs_line** is non-zero (see *systune*(1M)).

FILES

/usr/include/sys/param.h
/usr/include/sys/types.h
/usr/include/sys/inode.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), efs(4), types(5).

NAME

lvtab – information about logical volumes

DESCRIPTION

The file */etc/lvtab* describes the logical volumes used by the local machine. There is an entry in this file for every logical volume which is used by the system. It is read by commands that create, install and check the consistency of logical volumes. The system administrator can modify it with a text editor to add new logical volumes or to extend existing ones.

The file consists of entries which have the form:

volume_device_name:[volume_name]:[options:]device_pathnames

For example:

```
lv0:logical volume test:stripes=3:devs=/dev/dsk/ips0d1s7, \
/dev/dsk/ips0d2s7, /dev/dsk/ips0d3s7
```

Fields are separated by colons, and lines can be continued by the usual backslash convention as illustrated above. A '#' as the first non-white character indicates a comment; blank lines can be present in the file and are ignored.

The fields in each entry have the following significance:

volume_device_name
: This indicates the names of the special files through which the system accesses the logical volume. In the above example, the entry *lv0* implies that the logical volume is accessed via the device special files /dev/dsk/lv0 and /dev/rdsk/lv0. Note that volume device names are expected to be of the form 'lv' followed by one or 2 digits; this is enforced by the logical volume utilities.

volume name
: This is a human-readable identifying name for the logical volume. The logical volume labels on the disks constituting a volume also carry a copy of the volume name, so utilities are able to check that the logical volume on the disks physically present is actually the volume expected by */etc/lvtab*.

  This field can be null (indicated by a second colon immediately following the one terminating the *volume_device_name* field). This is legal but deprecated, since in this case, no identity check of the logical volume can be done by the utilities.

options
: Some numerical options concerning the volume can appear. These are specified in the format "option_name=number:". There must be no space between the option_name, the '=' sign, the numerical value given, and the terminating colon. Note that since the number of options is variable, the terminating colon is considered part of the option entry: it is not necessary to indicate omitted options.

Currently recognized options are:

stripes=
step=

The **stripes** option allows a striped logical volume to be created; the value of the parameter specifies the number of ways the volume storage is striped across its constituent devices. If this option is omitted, the logical volume is unstriped.

The **step** option is meaningful only for striped volumes (and is ignored otherwise); it specifies the granularity with which the storage is to be round-robin distributed over the constituent devices. If this option is omitted, the default **step** value is the device tracksize; this is generally a good value so the **step** option is not normally needed. **step** is in units of 512-byte blocks.

device_pathnames
Following any numerical options, there must be a list of the block special file pathnames of the devices constituting the logical volume. This is introduced by the keyword

devs=

The pathnames must be comma-separated.

Each pathname should be the name of the special file for a disk device partition in the */dev/dsk* directory. The partition must be one which is legal for use as normal data storage--it must not be one of the dedicated partitions such as the disk volume label, track replacement area, and so on.

If the volume is striped, some restrictions apply: the number of pathnames must be a multiple of **stripes**. Further, considering the pathnames as successive groups, each of **stripes** pathnames, the devices in each group must be all of the same size.

To obtain best performance from striping, each disk (within every group of **stripes** disks) should be on a separate controller.

The entries from this file are accessed using the routines in *getlvent*(3C), which returns a structure of the following form:

```
struct lvtabent     {
     char      *devname;      /* volume device name */
     char      *volname;      /* volume name (human-readable) */
     unsigned  stripe;        /* number of ways striped */
     unsigned  gran;          /* granularity of striping(step value)*/
     unsigned  ndevs;         /* number of constituent devices */
     int       mindex;        /* not currently used */
     char      *pathnames[1]; /* pathnames of constituent devices */
};
```

This structure is defined in the <*lvtab.h*> include file.

FILES

/etc/lvtab

SEE ALSO

lvck(1M), lvinit(1M), mklv(1M), getlvent(3C), lv(7M).

NAME

master – master configuration database

DESCRIPTION

The *master* configuration database is a collection of files. Each file contains configuration information for a device or module that can be included in the system. A file is named with the module name to which it applies. This collection of files is maintained in a directory called */var/sysgen/master.d*. Each individual file has an identical format. For convenience, this collection of files is referred to as the *master* file, as though it was a single file. This allows a reference to the *master* file to be understood to mean the *individual file* in the *master.d* directory that corresponds to the name of a device or module.

The *master* file is used by the *lboot*(1M) program to obtain device information to generate the device driver and configurable module files. *master* consists of two parts; they are separated by a line with a dollar sign ($) in column 1. Part 1 contains device information for both hardware and software devices and loadable modules. Part 2 contains parameter declarations. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1, Description

Hardware devices, software drivers, and loadable modules are defined with a line containing the following information. Field 1 must begin in the leftmost position on the line. Fields are separated by white space (tab or blank).

Field 1: Element characteristics:

- **o** specify only once
- **r** required device
- **b** block device
- **c** character device
- **t** initialize cdevsw[].d_ttys
- **j** filesystem
- **s** software driver
- **f** STREAMS driver
- **m** STREAMS module
- **x** not a driver; a loadable module
- **k** kernel module
- **u** a stubs module that is loaded after all other normal modules
- **n** driver is fully semaphored for multi-processor operation; the **n** and **p** directives are ignored on single-processor systems
- **p** driver is not semaphored and should run on only one processor
- **w** driver is prepared to perform any cache write back operation required on write data passed via the strategy routine
- **d** dynamically loadable kernel module

R auto-registrable dynamically loadable kernel module
N don't allow auto-unload of dynamically loadable kernel module
D load, then unload a dynamically loadable kernel module
e ethernet driver

Field 2: Handler prefix (14 characters maximum).

Field 3: Software driver external major number; a dash (–) if not a software driver or to be assigned during execution of *lboot*(1M). Multiple major numbers can be specified, separated by commas.

Field 4: Number of sub-devices per device; a dash (–) if none.

Field 5: Dependency list (optional); this is a comma-separated list of other drivers or modules that must be present in the configuration if this module is to be included

For each module, two classes of information are required by *lboot*(1M): external routine references and variable definitions. Routine lines begin with white space and immediately follow the initial module specification line. These lines are free form, thus they can be continued arbitrarily between non-blank tokens as long as the first character of a line is white space. Variable definition lines begin after a line that contains a $ in column one. Variable definitions follow C language conventions, with slight modifications.

Part 1, Routine Reference Lines

If the IRIX system kernel or other dependent module contains external references to a module, but the module is not configured, these external references are undefined. Therefore, the routine reference lines are used to provide the information necessary to generate appropriate dummy functions at boot time when the driver is not loaded.

Routine references are defined as follows:

Field 1: Routine name ()

Field 2: The routine type; one of

{}	routine_name(){}
{nulldev}	routine_name(){nulldev();}
{nosys}	routine_name(){return nosys();}
{nodev}	routine_name(){return nodev();}
{false}	routine_name(){return 0;}
{true}	routine_name(){return 1;}
{fsnull}	routine_name(){return fsnull();}
{fsstray}	routine_name(){return fsstray();}
{nopkg}	routine_name(){nopkg();}

{noreach} routine_name(){noreach();}

Part 2, Variables

Variables can be declared and (optionally) statically initialized on lines after a line whose first character is a dollar sign ($). Variable definitions follow standard C syntax for global declarations, with the following inline substitutions:

##M The internal major number assigned to the current module if it is a device driver; zero if this module is not a device driver.

##E The external major number assigned to the current module; either explicitly defined by the current master file entry, or assigned by *lboot*(1M).

##C The number of controllers present; this number is determined dynamically by lboot(1M) for hardware devices, or by the number provided in the system file for non-hardware drivers or modules.

##D The number of devices per controller taken directly from the current master file entry.

EXAMPLES

A sample *master* file for a shared memory module is named *shm*. The module is an optional loadable software module that can only be specified once. The module prefix is *shm*, and it has no major number associated with it. In addition, another module named *ipc* is necessary for the correct operation of this module.

```
*FLAG PREFIX SOFT #DEV DEPENDENCIES
ox    shm    –     –    ipc
               shmsys(){nosys}
               shmexec(){}
               shmexit(){}
               shmfork(){}
               shmslp(){true}
               shmtext(){}
$
#define SHMMAX 131072
#define SHMMIN 1
#define SHMMNI 100
#define SHMSEG 6
#define SHMALL 512

struct shmid_ds shmem[SHMMNI];
struct shminfo shminfo = {
     SHMMAX,
     SHMMIN,
```

```
        SHMMNI,
        SHMSEG,
        SHMALL,
};
```

This *master* file causes routines named *shmsys*, *shmexec*, and so on to be generated by the boot program if the *shm* driver is not loaded and there is a reference to this routine from any other module loaded. When the driver is loaded, the structure array *shmem* is allocated, and the structure *shminfo* is allocated and initialized as specified.

A sample *master* file for a VME disk driver is named *dkip*. The driver is a block and a character device, the driver prefix is *dkip*, and the external major number is 4. The VME interrupt priority level and vector numbers are declared in the system file */var/sysgen/system* (see *lboot*(1M)).

```
*FLAG PREFIX SOFT #DEV DEPENDENCIES
bc    dkip   4    -    io

$$$
/* disk driver variable tables */
#include "sys/dvh.h"
#include "sys/dkipreg.h"
#include "sys/elog.h"

struct iotime dkipiotime[##C][DKIPUPC]; /* io statistics */
struct iobuf dkipctab[##C];             /* controller queues */
struct iobuf dkiputab[##C][DKIPUPC];    /* drive queues */
int dkipmajor = ##E;                    /* external major # */
```

This *master* file causes entries in the block and character device switch tables to be generated if this module is loaded. Since this is a hardware device (implied by the block and character flags), VME interrupt structures are also generated by the boot program. The declared arrays are all sized to the number of controllers present, which is determined by the boot program based on information in the system file */var/sysgen/system*.

FILES

/var/sysgen/master.d/*
/var/sysgen/system

SEE ALSO

lboot(1M), mload(4), system(4).

NAME

mload – dynamically loadable kernel modules

DESCRIPTION

IRIX supports dynamic loading and unloading of modules into a running kernel. Kernel modules can be registered and then loaded automatically by the kernel when the corresponding device is opened, or they can be loaded manually. Similarly, dynamically loaded modules can be unloaded automatically or manually if the module includes an **unload** entry point. A loadable kernel module can be a character, block or streams device driver, a streams module, a library module or the *idbg.o* module.

Module Configuration

Each loadable module should contain the string:

```
char *prefixmversion = M_VERSION;
```

M_VERSION is defined in the *mload.h* header file, which should be included by the loadable module.

A loadable module must be compiled with the following *cc* options (use **uname −s** to determine which set of options to use; if **IRIX64** is printed, use the 64-bit set, otherwise use the 32-bit set):

For 32-bit modules: **−non_shared −elf −G0 −Wc,−pic0 −r −d −c −jalr**

For 64-bit modules: **−c −non_shared −elf −G0 −jalr −64 −mips3**

−non_shared
Produce a static executable. The output object created will not use any shared objects during execution.

−elf Produce an ELF object.

−G0 Disable global pointer since it is not supported for loadable modules. For more information about the global pointer, refer to *gp_overflow*(5).

−Wc,−pic0 Do not allocate extra stack space which is not necessary for non_shared objects.

−r Retain relocation entries in the output file.

−d Force definition of common storage and define loader defined symbols. Without this option, space is not allocated in bss for common variables.

−c Suppress the loading phase of the compilation and force an object file to be produced even if only one program is compiled.

–jalr Force the compiler to produce jalr instructions rather than jal instructions. A jal instruction has a 26 bit target, so if a module is loaded into K2SEG, for example, it could not call a kernel routine in K0SEG.

–64 Produce a 64-bit object.

–mips3 Produce code using the full MIPS III (R4000) instruction set.

A loadable module must not be dependent on any loadable module, other than a library module. In order to load a module comprised of multiple object files, the object files should be linked together into a single object file, using the following *ld* options:

For 32-bit modules: **–non_shared –elf –G0 –r –d**

For 64-bit modules: **–non_shared –elf –G 0 –r –d**

Loading a Dynamically Loadable Kernel Module

Either *lboot* or the *ml* command can be used to load, register, unload, unregister, and list loadable kernel modules. The *lboot* command parses module type, prefix and major number information from the module's master file found in the */var/sysgen/master.d* directory. The loadable object file is expected to be found in the */var/sysgen/boot* directory. The *ml* command also provides a means of loading, registering and unloading loadable modules, without the need for creating a master file or reconfiguring the kernel.

Load

When a module is loaded, the object file's header is read; memory is allocated for the module's text, data and bss; the module's text and data are read; the module's text and data are relocated and unresolved references into the kernel are resolved; a symbol table is created for the module; the module is added to the appropriate kernel switch table; and the module's init routine is called.

A module is loaded using the following *ml* command:

ml ld [**-v**] **-**[**cbBfmi**] *module.o* **-p** *prefix* [**-s** *major major* ...]
[**-a** *modname*]

If a module is loaded successfully, an id number is returned which can be used to unload the module.

A module can also be loaded using *lboot*:

lboot -L *master*

Register
The register command is used to register a module for loading when its corresponding device is opened. When a module is registered, a stub routine is entered into the appropriate kernel switch table. When the corresponding device is opened, the module is actually loaded.

A module is registered using the following *ml* command:

ml reg [**-v**] **-**[**cbBfmi**] *module.o* **-p** *prefix* [**-s** *major major* ...]
[**-a** *modname*] [**-t** *autounload_delay*]

If a module is registered successfully, an id number is returned which can be used to unregister the module.

A module can also be registered using *lboot*:

lboot -R *master*

Unload
A module can be unloaded only if it provides an **unload** entry point. A module is unloaded using:

ml unld *id* [*id id* ...]

or

lboot -U *id* [*id id* ...]

Unregister
A module can be unregistered using:

ml unreg *id* [*id id* ...]

or

lboot -W *id* [*id id* ...]

List All loaded and/or registered modules can be listed using:

```
ml list[-rlb]
```

or

```
lboot -V
```

Master File Configuration

If a dynamically loadable module has an associated master file, the master file should include a **d** in Field 1. The **d** flag indicates to *lboot* that the module is a dynamically loadable kernel module. If the **d** flag is present *lboot* will parse the module's master file, but will not fill in the entry in the corresponding kernel switch table for the module. All global data defined in the master file will be included in the generated *master.c* file. The kernel should be configured with master files that contain the **d** option for each module that will be a dynamically loadable module, if *lboot* will be used to load, register, unload, unregister or autoregister the module. If the *ml*(1M) command will be used, then it is not necessary to create a master file for the module.

Auto Registration

Loadable modules can be registered by *lboot* automatically at system startup when autoconfig is run. In order for a module to be auto-registered, its master file should contain an **R** in Field 1, in addition to **d**, which indicates that the module is loadable. When *lboot* runs at system startup, it registers each module that contains an **R** in its master file. Modules which specify a particular major number are registered before modules which pick a major number dynamically. If an rc2 script is added, which registers or loads dynamically loadable modules, it should be run after the autoconfig rc2 script is run.

For more detailed information, see the *lboot*(1M), *ml*(1M), and *master*(4) reference pages.

Auto Unload

All registered modules that include an **unload** routine are automatically unloaded after last close, unless they have been configured not to. Modules are unloaded five minutes after last close by default. The default auto-unload delay can be changed by using *systune* to change the *module_unld_delay* variable. For more information about systune, see the *systune*(1M) reference page. A particular module can be configured with a specific auto-unload delay by using the *ml* command. A module can be configured to not be auto-unloaded by either placing an **N** in the flags field of its *master.d* file, if it is registered using *lboot*, or by using *ml* to register the module and using the **–t** option.

Kernel Configuration

A kernel which supports loadable modules, should be configured so that the kernel switch tables generated by *lboot*(1M) contain "extra" entries for the loadable modules. Extra entries are generated by *lboot* based on the values of the following kernel tuneable parameters:

* name	default	minimum	maximum
bdevsw_extra	21	1	254
cdevsw_extra	23	3	254
fmodsw_extra	20	0	
vfssw_extra	5	0	

These tuneable parameters are found in the kernel */var/sysgen/mtune/kernel* file and are set to the defaults listed above. For more information about changing tuneable parameters, see the *mtune*(4) and *systune*(1M) reference pages.

Module Entry Points

Loadable device drivers should conform to the SVR4 DDI/DKI standard. In addition to the entry points specified by the DDI/DKI standard, if a loadable module is to be unloaded, the module needs to contain an **unload** entry point:

```
int prefixunload (void)
```

An **unload** routine should be treated as an interrupt routine and should not call any routines that would cause it to sleep, such as: **biowait**(), **sleep**(), **psema**() or **delay**().

An **unload** routine should free any resources allocated by the driver, including freeing interrupt vectors and allocated memory and return 0.

Module Initialization

After a module is loaded, linked into the kernel and sanity checking is done, the modules' initialization routines, *prefix***init**(), *prefix***edtinit**() and *prefix***start**() are called, if they exist. For more information on these routines, refer to the SVR4 DDI/DKI Reference Manual and the *IRIX Device Driver Programmer's Guide*.

Edt Type Drivers

For drivers that have an edtinit entry point, which get passed a pointer to an edt structure, *lboot* must be used to load the driver. A vector line should be added to the system file for the driver, as it would for any driver. When the module is loaded, using *lboot*, *lboot* parses the vector line from the system file to create an edt structure which is passed through the kernel and to the driver's edtinit routine. For more information, see the *system*(4) reference page.

Library Modules

A library module is a loadable module which contains a collection of functions and data that other loaded modules can link against. A library module can be loaded using the following *ml* command:

```
ml ld [-v] -l library.o
```

A library module must be loaded before other modules that link against it are loaded. Library modules can not be unloaded, registered or unregistered. Only regular object files are supported as loadable library modules.

The idbg.o Module

The *idbg.o* module can be dynamically loaded into a running kernel, so that the kernel print utility, *idbg*(1M), can be used without reconfiguring and rebooting a new kernel. The *idbg.o* module can be dynamically loaded using the *ml* command:

```
ml ld -i /var/sysgen/boot/idbg.o
```

The *idbg.o* module can also be unloaded.

Other idbg modules, such as *xfsidbg.o*, *xlvidbg.o*, *mloadidbg.o*, and so on, can be loaded after *idbg.o* is loaded. For example:

```
ml ld -i /var/sysgen/boot/xfsidbg.o -p xfsidbg.o
```

For more information, see the *idbg*(1M) reference page.

Loadable Modules and Hardware Inventory

Many device drivers add to the hardware inventory in their init or edtinit routines. If a driver is a dynamically loadable driver and is auto-registered, it will not show up in the hardware inventory until the driver has been loaded on the first open of the corresponding device. If a clean install or a diskless install is done, a */dev* entry will not get created by *MAKEDEV* for such a driver since it doesn't appear in the hardware inventory. If such a situation arises, the **D** *master.d* flag can be used to indicate that the driver should be loaded, then unloaded by *autoconfig*. If the **R** *master.d* flag, which indicates that the driver should be auto-registered, is also used, then the driver will be auto-registered as usual. A startup script can then be added that will run *MAKEDEV* after *autoconfig*, if necessary. For an example, see the */etc/init.d/chkdev* startup script.

Kernel Runtime Symbol Table

A runtime symbol table which contains kernel routines and global data that modules can link against is created from the ELF symbol information in the kernel that was booted. The runtime symbol table is created automatically by the kernel from the file indicated by the **kernname** environment variable, which is set by sash to the name of the file that was booted.

The symbol table is loaded with a default auto-unload timeout of five minutes, after which the symbol table is automatically unloaded. The symbol table is automatically reloaded when needed to resolve symbols (for example when a new or registered module is loaded).

The kernel runtime symbol table can also be loaded manually, using the *ml* command:

```
ml ld -r /unix
```

Or unloaded manually:

ml unld *id*

Note that only one kernel runtime symbol table can exist at one time.

Auto-loading and unloading of the kernel runtime symbol table can be disabled using the **mload_auto_rtsyms** systune variable. For more information about tuneable variables, see the *systune*(1M) reference page.

Debugging Loadable Modules

symmon(1M) supports debugging of loadable modules. *symmon* commands that do a symbol table lookup, such as: brk, lkup, lkaddr, hx and nm, also search the symbol tables created for loadable modules. The *msyms* command can also be used to list the symbols for a particular loaded module:

msyms *id*

The *mlist* command can be used to list all of the modules that are currently loaded and/or registered.

For more information, see the *symmon*(1M) reference page.

Load/Register Failures

If a registered module fails to load, it is suggested that the module be unregistered and then loaded using **ml ld** or **lboot −L**, in order to get a more detailed error message about the failure. All of the error codes, including a description of each, are listed in the *mload.h* header file, found in the */usr/include/sys* directory.

The kernel will fail to load or register a module for any of the following reasons:

1. If autoconfig is not run at system startup, none of the dynamically loadable modules will be registered or loaded.
2. If autoconfig fails for some reason, before it has processed the dynamically loadable module *master.d* files, the modules will not be registered or loaded.
3. The major number specified either in the master file, or by the ml command, is already in use.
4. The object file is not compiled with the correct options, such as **−G0** and **−jalr**.
5. The module is an "old style" driver, with either xxxdevflag set to D_OLD, or no xxxdevflag exists in the driver.
6. A corrupted object file could cause "invalid JMPADDR" errors.
7. Not all of the module's symbols were resolved by the kernel.

8. The device switch table is full and has no more room to add a loadable driver.

9. Required entry points for the particular type of module are not found in the object file, such as xxxopen for a character device driver.

10. All major numbers are in use.

11. An old sash is used to boot the kernel, which does not set the **kernname** environment variable, which indicates the on-disk kernel image to load the runtime symbol table from (for example, */unix*). This will cause all loadable modules to fail to load or be registered. To find out what the **kernname** environment variable is set to, use the *nvram*(1M) command:

```
nvram kernname
```

12. The runtime symbol table can not be loaded from the file indicated by the **kernname** environment variable, because the file does not exist, the file is not the same as the running kernel or the kernel was bootp'ed from another machine.

EXAMPLE 1

The following example lists the steps necessary to build a kernel and load a character device driver, called dlkm, using the *lboot* command:

1. Add **d** to the dlkm master file:

```
*FLAG   PREFIX  SOFT    #DEV    DEPENDENCIES
cd       dlkm     38      2
```

2. Make sure that the cdevsw_extra kernel tuneable parameter allows for extra entries in the cdevsw table, the default setting in */var/sysgen/mtune/kernel* is:

```
cdevsw_extra        23        3      254
```

The *systune*(1M) command also lists the current values of all of the tuneable parameters. If the kernel is not configured to allow extra entries in the cdevsw table, use the *systune* command to change the cdevsw_extra parameter:

```
> systune -i
systune-> cdevsw_extra 3
systune-> quit
>
```

3. Build a new kernel and boot the target system with the new kernel.

4. Compile the *dlkm.c* driver:

   For 32-bit modules:

   ```
   cc -non_shared -elf -G0 -r -d -jalr -c dlkm.c
   ```

   For 64-bit modules:

   ```
   cc -non_shared -elf -G 0 -jalr -c dlkm.c
   ```

5. Copy *dlkm.o* to */var/sysgen/boot*.

6. Load the driver into the kernel:

   ```
   lboot -L dlkm
   ```

7. List the currently loaded modules to verify that the module was loaded:

   ```
   lboot -V
   ```

EXAMPLE 2

The following example lists the steps necessary to load a character device driver, called dlkm, using the *ml* command:

1. Follow step 2 from example 1.

2. Follow step 4 from example 1.

3. Load the driver into the kernel:

   ```
   ml ld -c dlkm.o -p dlkm -s 38
   ```

   If a major number is not specified, the first free major number in the MAJOR table is used. If the load was successful, an id number is returned, which can be used to unload the driver.

4. List the currently loaded modules to verify that the module was loaded:

```
ml list
```

CAVEATS

1. Loadable modules must not have any dependencies on loadable modules, other than library modules. When a module is loaded, it is linked against the kernel symbol table and any loaded library modules' symbol tables, but it is not linked against other modules' symbol tables.

2. Only character, block and streams device drivers, streams modules and library modules are supported as loadable modules at this time.

3. Old style drivers (devflag set to D_OLD) are not loadable.

4. Kernel profiling does not support loadable modules.

5. Memory allocated may be in either K0SEG or in K2SEG. If the module is loaded into K2SEG static buffers are not necessarily in physically contiguous memory.

SEE ALSO

cc(1), lboot(1M), ld(1), ml(1M), symmon(1M), systune(1M), master(4), mtune(4).

IRIX Device Driver Programmer's Guide

NAME

mtune – default system tunable parameters

DESCRIPTION

The directory **/var/sysgen/mtune** contains information about all the system tunable parameters, including default values. The files in this directory should never be changed. Instead, use the **systune**(1M) utility to change parameters in the **/var/sysgen/stune** file.

Each loadable module can have its own **mtune** file, which is placed in the **mtune** directory and has the same name as the module. Parameters in an **mtune** file may be grouped together in groups, according to the nature of the parameters. For example, all parameters dealing with the number of processes that can run on the system at any given time are grouped together in the **numproc** group in the **kernel** module. The syntax of an **mtune** module file is given below:

```
[<group name>: [<flag>]]
<parameter clauses>
```

Names that end with a colon character, **:**, are group names. Parameters can be grouped together in groups so that one sanity checking function can be used to verify the values and the dependencies between these variables. The group name is optional if there is only one group in the module. For this case, the configuration tools use the module name as the group name.

The group name is followed by a flag. The flag can be either **run** or **static**. If the flag is **run**, this group of tunable variables can be changed with the command **systune** on a running system. Otherwise, the variables are set at initialization time and can be changed only by creating a new kernel and booting that kernel. Modules with no group specifier or a group specifier without a flag default to **static**.

Each tunable parameter is specified by a single line, a parameter clause, in the file. Blank lines and lines beginning with **#** or ***** are considered comments and are ignored. The syntax for each line is:

```
<name>[,<tag>] <default value> [[<min value> [<max value> [ll|LL]]
```

<name> The name of the tunable parameter. It is used to pass the value to the system when a kernel is built or changed by **systune** command. Since this name is made into a global variable name, using a long descriptive name is useful to avoid any name collisions.

<tag> This optional field is separated from <name> by a comma. It is used to qualify whether the tunable parameter should be used in the configuration being built. This allows a single **tune** file to be used in multiple different configurations. Parameters without any <tag> are always used, those with a <tag> are only used if the tag matches one of the <tag>s specified in the **system** file (see **system**(4)). Only one <tag> is permitted on a given line.

<default value> The default value of the tunable parameter. If the value is not specified in the **stune** file, this value is used when the system is built. This value is mandatory.

<min value> The minimum allowable value for the tunable parameter. If the parameter is set in the **stune** file, the **lboot** command checks that the new value is equal to or greater than this value. The command **systune** also verifies the new value against this value before changing the system. This field is optional; a value of 0 is equivalent to not specifying a value.

<max value> The maximum allowable value for the tunable parameter. If the parameter is set in the **stune** file, the **lboot** command checks that the new value is equal to or less than this value. The command **systune** also verifies the new value against this value before changing the system. This field is optional; a value of 0 is equivalent to not specifying a value.

ll|**LL** By default, each tunable parameter is represented by an global variable in the kernel of type int (32 bits). Some tunable parameters may need to be specified as 64 bit quantities. Adding an **ll** or **LL** to the end of the parameter specification causes **lboot** to represent the parameter as a **long long**.

FILES

/var/sysgen/mtune/*	default system tunable parameters
/var/sysgen/stune	local settings for system tunable parameters
/var/sysgen/system/*	master system configuration files

SEE ALSO

lboot(1M), systune(1M), stune(4), system(4).

NAME

passwd – password file

DESCRIPTION

/etc/passwd is an ASCII file containing entries for each user. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a newline. An entry beginning with # is ignored.

The *passwd* file contains the following information for each user:

name — User's login name — consists of alphanumeric characters and must not be greater than eight characters long. It is recommended that the login name consist of a leading lower case letter followed by a combination of digits and lower case letters for greatest portability across multiple versions of the UNIX operating system. This recommendation can be safely ignored for users local to IRIX systems. The *pwck*(1M) command checks for the greatest possible portability on names, and complains about user names that do not cause problems on IRIX.

password — Encrypted password and optional password aging information. If the password field is null (empty), no password is demanded when the user logs in. If the system is configured to use shadow passwords, this field of */etc/passwd* is ignored by all programs that do password checking. See *pwconv*(1M) for information about shadow passwords.

numerical user ID
: This is the user's ID in the system and it must be unique.

numerical group ID
: This is the number of the group that the user belongs to.

user's real name
: In some versions of UNIX, this field also contains the user's office, extension, home phone, and so on. For historical reasons this field is called the GECOS field. The *finger*(1) program can interpret the GECOS field if it contains comma (‘‘,’’) separated subfields as follows:

name	user's full name
office	user's office number
wphone	user's work phone number
hphone	user's home phone number

  An & in the user's full name field stands for the login name (in cases where the login name appears in a user's real name).

initial working directory
: The directory that the user is positioned in when they log in; this is known as the home directory.

shell The program to use as the command interpreter (shell) when the user logs in. If the *shell* field is empty, the Bourne shell (*/bin/sh*) is assumed. If the first character of this field is an *, then the *login*(1) program treats the home directory field as the directory to be used as the argument to the *chroot*(2) system call, and then loops back to reading the */etc/passwd* file under the new root, reprompting for the login. This can be used to implement secure or restricted logins, in a manner similar to *ftp*(1C).

Password aging is used for a particular user if his encrypted password is followed by a comma and a non-null string of characters from a 64-character alphabet (.,/,0-9, A-Z, a-z). The first character of the age, *M* say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired is forced to change his password. The next character, *m* say, denotes the minimum period in weeks that must expire before the password can be changed. If the second character is omitted, zero weeks is the default minimum. *M* and *m* have numerical values in the range 0–63 that correspond to the 64-character alphabet shown above (**/** = 1 week, **z** = 63 weeks). If *m* = *M* = 0 (derived from the string **.** or **..**) the user is forced to change his password the next time he logs in (and the age disappears from his entry in the password file). If *m* > *M* (signified, for example, by the string **./**), only the superuser is able to change the password.

The password file resides in the */etc* directory. Because of the encrypted passwords, it has general read permission and can be used, for example, to map numerical user ID's to names.

NIS ENTRIES

If the NFS option is installed, the *passwd* file can also have lines beginning with a '**+**' (plus sign) which means to incorporate entries from the NIS. There are three styles of **+** entries in this file:

+ Means to insert the entire contents of the NIS password file at that point.

+name Means to insert the entry (if any) for *name* from the NIS at that point.

+@netgroup Means to insert the entries for all members of the network group *netgroup* at that point.

If a + entry has a non-empty password, directory, GECOS, or shell field, the value of that field overrides what is contained in the NIS. The *uid* and *gid* fields cannot be overridden.

The *passwd* file can also have lines beginning with a '–' (minus sign) which means to disallow entries from the NIS. There are two styles of '–' entries in this file:

–name Means to disallow any subsequent entries (if any) for *name* (in this file or in the NIS).

–@netgroup Means to disallow any subsequent entries for all members of the network group *netgroup*.

Password aging is not supported for NIS entries.

UID CONVENTIONS

User ID number restrictions and conventions in the UNIX community are few and simple.

Reserved:

UID 0 — The superuser (aka root).

UID –1 — Invalid UID. Used to pass your ID on a 'wire' to remote systems. See system calls like *chmod*(2).

UID –2 — NFS 'nobody'. Note that because uid_t is unsigned, -2 is mapped to the special value 60001 by NFS.

UID 60001 and 60002
: For historical reasons, these values correspond to the users ''nobody'' and ''noaccess'', respectively. It is recommended that you not allocate these values to real users.

Conventions:

UID 1 to 10 — Commonly used for system pseudo users and daemons.

UID 11 to 99 — Commonly used for uucp logins and 'famous users'.

UID 100 to 2147483647 (except for 60001 and 60002)
: Normal users (start at 100). For historical reasons certain operations are restricted for uids larger than 65535. Most significantly, these users cannot own files on an *efs*(4) filesystem. This also means that they cannot run a program that allocates a *pty*(7M) (for example, *vi*(1) and *xwsh*(1G)) if */dev* resides on an *efs*(4) filesystem.

  For these reasons, we recommend that large uids only be used on *xfs*(4) based systems.

EXAMPLE

Here is a sample */etc/passwd* file:

```
root:q.mJzTnu8icF.:0:10:superuser:/:/bin/csh
bill:6k/7KCFRPNVXg,z/:508:10:& The Cat:/usr2/bill:/bin/csh
+john:
+@documentation:no-login:
+::::Guest
nobody:*:-2:-2::/dev/null:/dev/null
```

In this example, there are specific entries for users *root* and *bill*, to assure that they can log in even when the system is running stand-alone or when the NIS is not running. The user *bill* has 63 weeks of maximum password aging and 1 week of minimum password aging. Programs that use the GECOS field replace the

& with 'Bill'. The user *john* has his password entry in the NIS incorporated without change; anyone in the netgroup *documentation* has their password field disabled, and anyone else is able to log in with their usual password, shell, and home directory, but with a GECOS field of *Guest.* The user *nobody* cannot log in and is used by the *exportfs*(1M) command.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), pwck(1M), pwconv(1M), ypchpass(1), yppasswd(1), a64l(3C), crypt(3C), getpwent(3C), exports(4), group(4), netgroup(4), shadow(4).

NAME

profile – setting up an environment at login time

SYNOPSIS

/etc/profile
$HOME/.profile

DESCRIPTION

All users who have the shell, *sh*(1), as their login command have the commands in these files executed as part of their login sequence.

/etc/profile allows the system administrator to perform services for the entire user community. Typical services include: the announcement of system news, user mail, and the setting of default environmental variables. It is not unusual for */etc/profile* to execute special actions for the **root** login or the *su*(1M) command.

The file *$HOME/.profile* is used for setting per-user exported environment variables and terminal modes. The following example is typical (except for the comments):

```
#  Set the file creation mask to prohibit
#  others from reading my files.
umask 027
#  Add my own /bin directory to the shell search sequence.
PATH=$PATH:$HOME/bin
#  Set terminal type
eval ‘tset -S -Q‘
#  Set the interrupt character to control-c.
stty intr ^c
#  List directories in columns if standard out is a terminal.
ls()    { if [ -t ]; then /bin/ls -C $*; else /bin/ls $*; fi }
```

FILES

/etc/TIMEZONE	timezone environment
$HOME/.profile	user-specific environment
/etc/profile	system-wide environment

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1M), tput(1), tset(1), terminfo(4), timezone(4), environ(5), term(5).

NOTES

Care must be taken in providing system-wide services in */etc/profile*. Personal *.profile* files are better for serving all but the most global needs.

NAME

shadow – shadow password file

DESCRIPTION

/etc/shadow is an access-restricted ASCII system file. The fields for each user entry are separated by colons. Each user is separated from the next by a newline. Unlike the **/etc/passwd** file, **/etc/shadow** does not have general read permission. To create **/etc/shadow** from **/etc/passwd** use the **pwconv** command (see **pwconv**(1M)).

Here are the fields in **/etc/shadow**:

username The user's login name (ID).

password A 13-character encrypted password for the user, a *lock* string to indicate that the login is not accessible, or no string to show that there is no password for the login.

lastchanged The number of days between January 1, 1970 and the date that the password was last modified.

minimum The minimum number of days required between password changes.

maximum The maximum number of days the password is valid.

warn The number of days before that password expires that the user is warned.

inactive The number of days of inactivity allowed for that user.

expire An absolute date specifying when the login can no longer be used.

flag Reserved for future use; set to zero. Currently not used.

The encrypted password consists of 13 characters chosen from a 64-character alphabet (**.**, **/**, **0**–**9**, **A**–**Z**, **a**–**z**).

To update this file, use the **passwd** command.

FILES

/etc/shadow

SEE ALSO

login(1), passmgmt(1M), passwd(1), pwconv(1M), getspent(3C), putspent(3C), passwd(4).

NOTES

Shadow passwords can be used with NIS entries. If the shadow password file is present, each NIS entry must have a distinct shadow password entry, and the NIS-supplied encrypted password is not used. This effectively precludes the use of the NIS wildcard entry, +::-1:-1::: or netgroup (+@) expansions.

NAME

stune – local settings for system tunable parameters

DESCRIPTION

The file **/var/sysgen/stune** contains local system settings for tunable parameters. The parameter settings in this file replace the default values specified in **/var/sysgen/mtune/***, if the new values are within the legal range for the specified parameter. Blank lines and lines beginning with the # or * characters are considered comments and are ignored. The file contains one line for each parameter to be reset. The syntax for each line is:

parameter name[,*tag*] = *value* [**ll**|**LL**]

parameter name The name of the tunable parameter.

tag Optional field that if specified controls whether the parameter is used in the current configuration. Use of this field can permit multiple configurations to be present in a single **stune** file. See **mtune**(4) and **system**(4) for more details.

value The new value for the tunable parameter.

ll|**LL** This optional field specifies whether the parameter should be interpreted as a 32 or 64 bit quantity.

The file **stune** normally resides in **/var/sysgen**. You can edit this file, as root, as you find necessary. However, it is suggested that you use the system tuning tool, **systune**(1M), instead of making changes directly to the **stune** file. **systune** makes specified changes in the **stune** file for you. You should never directly edit the default configuration files in the **mtune** directory.

FILES

/var/sysgen/mtune/* default system parameters
/var/sysgen/stune local settings for system tunable parameters

SEE ALSO

lboot(1M), systune(1M), mtune(4), system(4).

NAME

system – system configuration information directory

DESCRIPTION

This directory contains files (with the **.sm** suffix) that are used by the **lboot** program to obtain configuration information. These files generally contain information used to determine if specified hardware exists, a list of software drivers to include in the load, and the assignment of system devices such as *rootdev*, as well as instructions for manually overriding the drivers selected by the self-configuring boot process.

Each major subsystem can have its own configuration file, for example: **irix.sm** (base operating system configuration file), **gfx.sm** (graphics subsystem configuration file), and so forth. **lboot** logically concatenates all files in the *system* directory with the **.sm** suffix and processes the results.

The syntax of the system files is given below. The parser for the **/var/sysgen/system/*.sm** file is case sensitive. All uppercase strings in the syntax below should be uppercase in the **/var/sysgen/system/*.sm** file as well. Nonterminal symbols are enclosed in angle brackets, <>, while optional arguments are enclosed in square brackets, []. Ellipses, ..., indicate optional repetition of the argument for that line.

<fname> ::= master filename from **/master.d** directory
<func> ::= interrupt function name
<device> ::= special device name | DEV(<major>,<minor>)
<major> ::= <number>
<minor> ::= <number>
<proc> ::= processor # as interpreted by **runon**(1)
<number> ::= decimal, octal or hex literal

lboot can determine if hardware exists for a given module by use of probe commands. The syntax for probe commands is:

<probe_cmd> ::= probe=<number>
[probe_size=<number>] | <extended_probe>
<extended_probe> ::= exprobe=<probe_sequence>
| exprobe=(<probe_sequence>,<probe_sequence>,...)
<probe_sequence> ::= (<seq>,<address>,<size>,<value>,<mask>)
<seq> ::= a sequence of 1 or more r's, rn's, or w's, indicating a
read from <address> or a write to <address>
<address> ::= <number>
<size> ::= <number>
<value> ::= <number>

<mask> ::= <number>

In order to deal with the high degree of configurability of the newer systems, a new complementary set of probe routines has been added which are used in conjunction with a new style of VECTOR line, described later in this reference page. The new probe commands are the only means to detect peripherals on the CHALLENGE and Onyx systems, but the new commands are supported on all Silicon Graphics platforms.

<probe_cmd> ::= probe_space=(<bus_space>,<number>
[probe_size=<number>] | <extended_probe>)
<extended_probe> ::= exprobe_space=<probe_sequence>
| exprobe_space=(<probe_sequence>,<probe_sequence>, ...)
<probe_sequence> ::= (<seq>,<bus_space>,<address>,<size>,
<value>,<mask>)
<seq> ::= a sequence of 1 or more r's, rn's, or w's, indicating a
read from <address>, or a write to <address>.
<bus_space> ::= A16NP | A16S | A24NP | A24S | A32NP | A32S
<address> ::= <number>
<size> ::= <number>
<value> ::= <number>
<mask> ::= <number>

As shown from the grammar, there are two forms of probe commands. The first allows the specification of an address to read, and optionally, a number of bytes to read. If a probe address is specified, the boot program attempts to read *probe_size* bytes (default 4) to determine if the hardware exists for the module. If the read succeeds, the hardware is assumed to exist, and the module is included.

The extended form specifies a sequence of one or more five-tuples used to determine if the hardware exists. Each five-tuple specifies a read/write *sequence*, an *address* to read or write, a *size* of up to four bytes, a *value*, and a *mask*. Then, for each five-tuple, the following is performed:

```
for each element in command do
     if element == 'w' then
          if write(address, value & mask, size) != size then
               failure
     if element == 'r' then
          if read(address, temp, size) != size then
               failure
          if suffix == 'n' then
               if temp & mask == value & mask then
                    failure
          else
               if temp & mask != value & mask then
                    failure
```

The lines listed below can appear in any order. Blank lines can be inserted at any point. Comment lines must begin with an asterisk. Entries for VECTOR, EXCLUDE, and INCLUDE are cumulative. For all other entries, the last line to appear in the file is used -- any earlier entries are ignored.

There are two styles of VECTOR line. The first version is the historical version and does not work on newer platforms such as the CHALLENGE and Onyx series. The second VECTOR command is the new version that supports the CHALLENGE and Onyx series along with newer bus types such as EISA. The second version is the preferred method since it works across all Silicon Graphics hardware platforms.

VECTOR: module=<fname> [intr=<func>]
[vector=<number> ipl=<number> unit=<number>] [base=<number>]
[base2=<number>] [base3=<number>]
[<probe_cmd>]
[intrcpu=<number>] [syscallcpu=<number>]

Specifies hardware to conditionally load. (Note that this must be a single line.) If a probe command is specified, the boot program performs the probe sequence, as discussed above. If the sequence succeeds, the module is included.

If a probe sequence is not specified, the hardware is assumed to exist. The **intr** function specifies the name of the module's interrupt handler. If it is not specified, the prefix defined in the module's master file (see **master**(4)) is concatenated with the string **intr**, and, if a routine with that name is found in the module's object (which resides in the directory **/var/sysgen/boot**), it is used as the interrupt routine.

If the triplet (vector, ipl, unit, base) is specified, a VME interrupt structure is assigned, using the corresponding VME address *vector*, priority level *ipl*, unit *unit*.

If the modules' object contains a routine whose name is the concatenation of the master file prefix and **edtinit**, that routine is involved once at startup and passed a pointer to an edt structure that contains the values for base, base2, base3, and a pointer to the VME interrupt structure.

If intrcpu is specified, it hints to the driver the desired CPU to take interrupts on. This is only a hint and may not be honored in all cases.

If syscallcpu is specified, it indicates the CPU to run non-MP driver syscalls on. This directive is always honored for non-MP drivers, and is silently ignored by MP drivers. This option should be used with caution because non-MP drivers may expect their syscalls and interrupts to run on the same CPU.

VECTOR: bustype=<bustype> module=<fname> adapter=<number> ipl=<number>
[intr=<func>] [vector=<number>] [ctlr=<number>]
[iospace=(<address-space>,<address>,<size>)]
[iospace2=(<address-space>,<address>,<size>)]
[iospace3=(<address-space>,<address>,<size>)]

[<probe_cmd>]

Specifies hardware to conditionally load. (Note that this must be a single line.) If a probe command is specified, the boot program performs the probe sequence, as discussed above. If the sequence succeeds, the module is included.

If a probe sequence is not specified, the hardware is assumed to exist. The bustype specifies the type of bus on which the device is connected. This is VME for a VME bus.

The adapter specifies to which bus of type bustype the device is connected. If adapter is set to *****, the system looks at each bus of type bustype to find the device.

The **intr** function specifies the name of the module's interrupt handler. If it is not specified, the prefix defined in the module's master file (see **master**(4)) is concatenated with the string **intr** and if a routine with that name is found in the module's object (which resides in the directory **/var/sysgen/boot**), it is used as the interrupt routine.

If the vector is not specified, it is assumed to be programmable. The ctlr field is used to pass a value into the driver that is specific to the device. This can be used to identify which device is present when there are multiple VECTOR lines for a particular device.

If the modules' object contains a routine whose name is the concatenation of the master file prefix and **edtinit**, that routine is involved once at startup and passed a pointer to an edt structure that contains the values for iospace, iospace2, iospace3, and a pointer to the bus info structure.

EXCLUDE: [<string>] ...

Specifies drivers to exclude from the load even if the device is found via VECTOR information.

INCLUDE: [<string>[(<number>)]] ...

Specifies software drivers or loadable modules to be included in the load. This is necessary to include the drivers for software devices. The optional <number> (parenthesis required) specifies the number of devices to be controlled by the driver (defaults to 1). This number corresponds to the builtin variable *##c* which can be referred to by expressions in part two of the **/var/sysgen/master** file.

ROOTDEV: <device>

Identifies the device containing the root filesystem.

SWAPDEV: <device> <number> <number>

Identifies the device to be used as swap space, the block number the swap space starts at, and the number of swap blocks available.

DUMPDEV: <device>

Identifies the device to be used for kernel dumps.

IPL: <IRQ level> <proc>

Send VME interrupt at <IRQ level> to <proc>. If <proc> does not exist at run time, the kernel defaults to use processor 0.

USE: [<string>[(<number>)] [<extended_probe>]] ...

If the driver is present, it is the same as INCLUDE. Behaves like EXCLUDE if the module or driver is not present in **/var/sysgen/boot**.

KERNEL: [<string>] ...

Specifies the module containing the heart of the operating system. It must be present in the system file.

NOINTR: <proc> ...

In CHALLENGE and Onyx systems, it provides a way to prevent processor(s) from receiving any interrupt other than the VME IRQ levels defined using IPL directive. This can be used for marking a processor for real time purpose. CPU 0 although should not be restricted from receiving interrupts. This directive is ignored on all other platforms.

LINKMODULES: <1|0>

If set to 1, this option causes **lboot** to ignore the **d** option in all master files and link all necessary modules into the kernel.

CC
LD

The names of the compiler and linker used to build the kernel. If absent, they default to **cc** and **ld**, respectively.

CCOPTS
LDOPTS

Option strings given to **cc**(1) and **ld**(1) respectively, to compile the **master.c** file and link the operating system.

TUNE-TAG: <string> ...

Sets a set of tags to be used to qualify the various tunable parameters for inclusion. If a tunable parameter has no tag (see **mtune**(4)), it is always included. If a tunable parameter has a tag, it is included only if the tag matches one of the tags specified by this parameter or via the **–O** option to **lboot**. Tags can be used to permit a single set of **mtune** and **stune** files to represent many different configurations.

FILES

/var/sysgen/system/*.sm
/usr/include/sys/edt.h

SEE ALSO

lboot(1M), master(4), mtune(4), stune(4).

NAME

sys_id – system identification (hostname) file

DESCRIPTION

The file */etc/sys_id* contains the name by which the system is known on communications networks such as the Internet and UUCP. The name can be up to 64 alphanumeric characters long and can include periods and hyphens. Periods are not part of the name but serve to separate components of a domain-style name. For example:

```
iris.widgets.com
```

During system startup this file is read by the script */etc/rc2.d/S20sysetup* and the contents are passed as a parameter to *hostname*(1) to initialize the system name. Once this has been done, this name is returned by the commands *hostname*(1) and *uname*(1) and by the system calls *gethostname*(2) and *uname*(2). *uname*(1) returns only the first eight characters up to the first period.

FILES

/etc/sys_id

SEE ALSO

hostname(1), uname(1), gethostname(2), uname(2), hostname(5).

NAME

ttytype – data base of terminal types by port

DESCRIPTION

ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to it. There is one line per port, containing the terminal kind (as a name described in *terminfo*(4)), a space, and the name of the tty, minus */dev/*.

This information is read by *tset*(1) to initialize the TERM environment variable at login time.

EXAMPLE

```
iris-ansi       console
vt100  ttyd1
?h19            ttyd2
?h19            ttyd3
?v50am          ttyd4
?v50am          ttyd5
?v50am          ttyd6
?v50am          ttyd7
?v50am          ttyd8
?v50am          ttyd9
?v50am          ttyd10
?v50am          ttyd11
?v50am          ttyd12
```

FILES

/etc/ttytype

SEE ALSO

tset(1), terminfo(4).

NAME

xfs – layout of the XFS filesystem

DESCRIPTION

An XFS filesystem can reside on a regular disk partition or on a logical volume (see *lv*(7M) and *xlv*(7M)). An XFS filesystem has up to three parts: a data section, a log section, and a real-time section. For disk partition and *lv* logical volume filesystems, the real-time section is absent, and the log area is contained within the data section. For XLV logical volume filesystems, the real-time section is optional, and the log section can be separate from the data section or contained within it. The filesystem sections are divided into a certain number of *blocks*, whose size is specified at *mkfs*(1M) time with the **–b** option.

The data section contains all the filesystem metadata (inodes, directories, indirect blocks) as well as the user file data for ordinary (non-real-time) files and the log area if the log is *internal* to the data section. The data section is divided into a number of *allocation groups*. The number and size of the allocation groups are chosen by *mkfs* so that there is normally a small number of equal-sized groups. The number of allocation groups controls the amount of parallelism available in file and block allocation. It should be increased from the default if there is sufficient memory and a lot of allocation activity. More allocation groups are added (of the original size) when *xfs_growfs*(1M) is run.

The log section (or area, if it is internal to the data section) is used to store changes to filesystem metadata while the filesystem is running until those changes are made to the data section. It is written sequentially during normal operation and read only during mount. When mounting a filesystem after a crash, the log is read to complete operations that were in progress at the time of the crash.

The real-time section is used to store the data of real-time files. These files had an attribute bit set through *fcntl*(2) after file creation, before any data was written to the file. The real-time section is divided into a number of *extents* of fixed size (specified at *mkfs* time). Each file in the real-time section has an extent size that is a multiple of the real-time section extent size.

Each allocation group contains several data structures. The first sector contains the superblock. For allocation groups after the first, the superblock is just a copy and is not updated after *mkfs*. The next three sectors contain information for block and inode allocation within the allocation group. Also contained within each allocation group are data structures to locate free blocks and inodes; these are located through the header structures.

Each XFS filesystem is labeled with a unique universal identifier (UUID). (See *uuid*(3C) for more details.) The UUID is stored in every allocation group header and is used to help distinguish one XFS filesystem from another, therefore you should avoid using *dd* or other block-by-block copying programs to copy XFS filesystems. If two XFS filesystems on the same machine have the UUID, *xfsdump* may become confused when doing incremental and resumed dumps. (See *xfsdump*(1M) for more details.) *xfs_copy* or *xfsdump*/*xfsrestore* are recommended for making copies of XFS filesystems.

All these data structures are subject to change, and the headers that specify their layout on disk are not provided.

SEE ALSO

attr(1), grio(1M), mkfs(1M), mkfs_xfs(1M), xfs_bmap(1M), xfs_check(1M), xfs_copy(1M), xfs_estimate(1M), xfs_growfs(1M), xfs_logprint(1M), xfsdump(1M), xfsrestore(1M), fcntl(2), syssgi(2), uuid(3C), filesystems(4), lv(7M), xlv(7M).

NAME

availmon – overview of system availability monitoring facilities

DESCRIPTION

The availability monitor (*availmon*) is a set of programs that collectively monitor and report the availability of a system and the diagnosis of system crashes. Using the monitor, it is possible to collect information such as the time a system became available (start-time), the duration for which it was available (up-time), the time it became unavailable (stop-time), the reason it became unavailable (stop-reason), the time it became available again (restart-time), and the duration for which it was unavailable (down-time).

The monitor differentiates controlled shutdowns (initiated by operators through *shutdown*(1M), *halt*(1M), and *init*(1M)), system panics (due to hardware errors or known fault points in kernel, for example, failed assertions), power failures, power cycles, and system resets (usually due to system hangs). If a memory dump is created and *icrash*(1M) is installed, the *icrash* report (including FRU analyzer report for high-end machines) is collected. Important syslog messages are also collected to help debugging. If FRU analyzer reports a problem related to hardware failure, the panic is determined to be due to hardware faults; otherwise, the panic is assumed to be due to software faults.

Controlled shutdowns can be initiated by an operator for a variety of reasons. From the point of view of classifying faults, these reasons can be grouped into two broad categories––**planned** and **abnormal**. For example, a shutdown to accommodate a pre-planned power outage is considered routine, as it does not reflect a fault in the system. On the other hand, a shutdown to replace faulty hardware is abnormal. At the time of shutdown (through *shutdown*(1M), *halt*(1M), and *init*(1M)), the operator is prompted to choose from one of the predefined set of reasons, so that *availmon* can classify the shutdown.

Information collected at a given system can be mailed (through internet) to concentrator accounts (that is, at Silicon Graphics) automatically, which can then maintain logs or databases of all systems being monitored. The default configuration, once enabled, sends information to Silicon Graphics Technical Support for entry into the Silicon Graphics availability database. Mailing of information does not compromise security; mail messages can be encoded. This encoding is safe to use in an international environment (it does not use DES, which is available only in the United States).

If a user chooses not to use internet mail to disseminate information, a concentrator account local to the user's network (possibly a special account at a server machine) can be chosen. Periodically, a system administrator can print the reports, and the reports can be security-cleared before being sent to Silicon Graphics.

CONFIGURATION

Once *availmon* is installed, the following configuration parameters can be used to control its operation.

autoemail The flag **autoemail** (see *amconfig*(1M)) controls the automatic emailing of the *availmon* reports. If the **autoemail** flag is **off**, only logging operations are performed; no email is sent to concentrator accounts. Default is **off**; **autoemail** is turned **on** by *amregister*(1M).

shutdownreason The flag **shutdownreason** (see *amconfig*(1M)) controls the shutdown reason query for controlled shutdowns. If the **shutdown** flag is **off**, no question is asked, and the shutdown reason is unknown. Default is **on** for high-end systems, **off** otherwise.

tickerd The flag **tickerd** (see *amconfig*(1M)) controls the ticker daemon that estimates system up time for system hangs. If the **tickerd** flag is **off**, the down time for system hangs is assumed to be one minute. Default is **on** for high-end systems, **off** otherwise.

autoemail.list The config file *autoemail.list* is used to specify lists of internet mail addresses of concentrator accounts for *availmon* reports and email formats. At least one white space character (blank, tab) should be used to separate two addresses in each list. The default *autoemail.list* specifies sending a diagnosis report to **availmon@csd.sgi.com** (compressed and encrypted).

REPORT VIEWING

The command *amreport*(1M) is provided to review *availmon* reports and to provide statistical availability information. This program can process local availability log files or received aggregate availability reports (site log file) from different systems.

amreport shows the statistical reports and availability reports hierarchically from overall statistics for all systems, a table of statistics for all systems (if the input is a local log file, the above information is not provided), statistics for each system, a table of all reboot instances for each system, to availability reports for each system. Please refer to *amreport*(1M) for details.

ADMINISTRATION EXAMPLES

Three examples are provided to illustrate the administration of *availmon*. The first (standard) example is for general customers that can send *availmon* reports to Silicon Graphics Technical Support and local Silicon Graphics support automatically. The second example is for secure sites that do not send out *availmon* reports automatically, but the system administrators would like to receive notice about system reboots and decide whether to send reports to Silicon Graphics. The reports can be filtered before sending. The third example is also for secure sites, except that no report is sent. System administrators need to check the system and process the reports themselves.

Standard Example

If *availmon* is installed on only one system, reboot the system after installation. Run *amregister* without any argument to register (turning on auto-email sends registration reports automatically) and configure the email lists. If the system is not IP19, IP21, IP22, or IP25, *amregister* asks to input system's serial number (from the back of the machine). **shutdownreason** and **tickerd** can be turned **on** or **off** anytime. The default *autoemail.list* is:

```
availability(compressed,encrypted):
availability(compressed):
availability(text):
diagnosis(compressed,encrypted): availmon@csd.sgi.com
```

```
diagnosis(compressed):
diagnosis(text):
```

In addition, the following may be desired:

```
availability(text): local_sysadmin
diagnosis(compressed,encrypted): local_SGI_support
```

If encrypted data in email is prohibited by law, move addresses in "(**compressed,encrypted**)" lists to "(**compressed**)" lists.

If *availmon* is installed on several systems, a site log file and automatic registration may be desired. After installing *availmon* and rebooting all systems, create an email alias in the system's aliases on the mail server or one system, pipeline availability reports to *amreceive*, and then append the output to a file. For example, if the site logfile is */disk/amrlog*, add one line to mail server's */etc/aliases*:

```
amrlog: "| /var/adm/avail/amreceive >> /disk/amrlog"
```

and run *newaliases*(1M) to set up this email alias. Then, run *amconfig* on one system to configure email lists. For example,

```
availability(compressed,encrypted):
availability(compressed):
availability(text): local_sysadmin amrlog
diagnosis(compressed,encrypted): availmon@csd.sgi.com local_SGI_support
diagnosis(compressed):
diagnosis(text):
```

Copy */var/adm/avail/config/autoemail.list* on this system to the rest of systems. Then, run **amregister −r** on all IP19, IP21, IP22, and IP25 systems. Writing a script and using rsh can automate this process. If **shutdownreason** or **tickerd** need to be turned **off** on some high-end systems (or on for some low-end systems), run **amconfig shutdownreason off (on)** or **amconfig tickerd off (on)** on those systems. For other platforms, run **amregister −r −s** *serial_number* to specify serial numbers, or run **amregister** without any option to input serial numbers. The serial numbers are important for Silicon Graphics Technical Support, so please provide this information. After everything is set up, **amreport −s /disk/amrlog** shows the overall statistics, system statistics, and individual availability reports.

Example for Secure Sites with Internal Report Mailing

The setup procedure is similar to Standard Example (excluding availmon@csd.sgi.com and *local_SGI_support* from *autoemail.list*. After system administrators receive *availmon* reports, they can check the latest diagnosis report, */var/adm/crash/diagreport*, on the system just rebooted, delete any sensitive data, and use *amsend* to mail the filtered report to availmon@csd.sgi.com and local Silicon Graphics support people. If the diagnosis report contains any ICRASH, SYSLOG, HINV, VERSIONS, or GFXINFO data, **amsend −i −z −x availmon@csd.sgi.com** *local_SGI_support* should be used to mail the report; if there is no such data in the report, use **amsend −d −z −x availmon@csd.sgi.com** *local_SGI_support*. If encrypted data

in email is prohibited by law, remove **–x** from the command.

Example for Secure Sites without Internal Report Mailing

Nothing special needs to be set up for this case. However, for those platforms not IP19, IP21, IP22, and IP25, *amregister* should still be run without any argument to input serial numbers and then to turn **off autoemail** so that reports generated by these systems are not automatically sent. **shutdownreason** and **tickerd** can also be turned **on** or **off**.

Since there is no report mailed after a system reboots, system administrators need to check if the system has been down and then evaluate the reports. If the system crashes more than once before checking, old reports are overwritten by the new one (core dumps and *icrash* reports are kept until being removed explicitly). Therefore, internal report mailing is recommended for secure sites.

Diagnosis reports can be sent to Silicon Graphics using *amsend* (check previous example for details). Another method is to run *amconfig* to configure standard email lists so that when reports need to be sent, *amnotify* can be used to send reports according to those lists.

FILES

/var/adm/avail/config/{autoemail,shutdownreason,tickerd}	configuration flag files
/var/adm/avail/config/autoemail.list	autoemail list configuration file
/var/adm/avail/availlog	primary log of availability monitor
/var/adm/avail/uptime	uptime in minutes (resolution of five minutes)
/etc/init.d/availmon	*init* script that logs start/stop and initiates notification

SEE ALSO

Mail(1), amconfig(1M), amnotify(1M), amparse(1M), amreceive(1M), amregister(1M), amreport(1M), amsend(1M), amtickerd(1M), chkconfig(1M), halt(1M), init(1M), shutdown(1M).

NAME

dks, jag – dksc (SCSI) disk driver

SYNOPSIS

/dev/dsk/dks*
/dev/dsk/jag*
/dev/rdsk/dks*
/dev/rdsk/jag*

DESCRIPTION

There can be up to seven SCSI drives attached per SCSI bus (15 for controllers that support wide SCSI), each of which can support a number of logical units (luns). The current limit is eight luns. Each unit (or lun) can have up to 16 partitions in use, three of which (8, 9, and 10) are special (see below).

Disk devices are named according to the following formats:

/dev/rdsk/dks*controller-*#**d***drive-*#{**s***partition-*#|**vh**|**vol**}
/dev/rdsk/dks*controller-*#**d***drive-*#**l***lun-*#{**s***partition-*#|**vh**|**vol**}
/dev/dsk/dks*controller-*#**d***drive-*#**s***partition-*#
/dev/dsk/dks*controller-*#**d***drive-*#**l***lun-*#**s***partition-*#
/dev/rdsk/jag*controller-*#**d***drive-*#{**s***partition-*#|**vh**|**vol**}
/dev/dsk/jag*controller-*#**d***drive-*#**s***partition-*#

The **rdsk** devices use a raw interface to communicate with the disk, while the **dsk** devices use a block interface. The *controller-*#, *drive-*#, and *lun-*# are used to indicate SCSI controller number, target ID, and logical unit number, respectively. **s***partition-*#, **vh**, and **vol** indicate a partition of the disk. The **vh** and **vol** devices are only in the **rdsk** directory, since they are normally used only for **ioctl** and raw access.

The standard partition allocation by Silicon Graphics has *root* on partition 0, *swap* on partition 1, and (optionally) */usr* on partition 6. Some systems, such as the Indy, are shipped from the factory with a single filesystem on the system disk for ease of administration. In this case, partition 6 is not used. Partition 7 (when present), normally maps the entire *usable* portion of the disk (excluding the volume header). Partition 8 (**vh**) maps the volume header (see *prtvtoc*(1M), *dvhtool*(1M)). Partition 10 (**vol**) maps the entire drive. Partition 9 is reserved, but is not used for disks with the **dksc** driver. Devices are not created automatically by **MAKEDEV** for the other partitions, and if used, must be created manually, or by modifying **MAKEDEV**.

The standard configuration has */dev/root* linked to partition 0 of the system disk, with */dev/swap* linked to partition 1 of the system disk, and */dev/usr* linked to partition 6 of the system disk. Systems that do not use partition 6 as shipped, instead using just the s0 and s1 partitions, still have a link made. There is no attempt to make the link to the device used for the **/usr** filesystem in the *fstab*(4) file, if present, even if it uses a different device. Option disks normally use the s7 partition as a single filesystem, containing the whole usable portion of the disk.

IOCTL FUNCTIONS

As well as normal read and write operations, the driver supports a number of special *ioctl*(2) operations when opened via the character special file in */dev/rdsk*. Command values for these are defined in the system include file <*sys/dkio.h*>, with data structures in <*sys/dksc.h*>.
These *ioctl* operations are intended for the use of special-purpose disk utilities. Many of them can have drastic or even fatal effects on disk operation if misused; they should be invoked only by the knowledgeable and with extreme caution!

A list of the *ioctl* commands supported by the *dks* driver is given below.

DIOCADDBB
: Adds a block to the badblock list. The argument is the logical block number (not a pointer) on the drive. For some makes of drives, the spared block must be written before the sparing takes effect. Only programs running with superuser permissions can use this ioctl.

DIOCDRIVETYPE
: The first **SCSI_DEVICE_NAME_SIZE** bytes (currently 28) of the SCSI inquiry data for the drive is returned to the caller. The argument is a pointer to a char array of at least this size. This contains vendor and drive specific information such as the drive name and model number. See a SCSI command specification for details on the structure of this buffer.

DIOCFORMAT
: Formats the entire drive. Any information on the drive is lost. The grown defect list (blocks spared with DIOCADDBB) is empty after formatting is complete, blocks previously in the grown defect list are no longer spared.

DIOCGETVH
: Reads the disk volume header from the driver into a buffer in the calling program. The argument in the ioctl call must point to a buffer of size at least 512 bytes. The structure of the volume header is defined in the include file <*sys/dvh.h*>. The corresponding call DIOCSETVH sets the drivers idea of the volume header; in particular, the drivers idea of the partition sizes and offsets is changed.

DIOCPREVREM
: Issues a PREVENT ALLOW MEDIA REMOVAL command to the opened device. The first bit of the arg is or'd into byte 4 of the SCSI command. See a SCSI command specification for details on this command.

DIOCRDEFECTS
: The argument is a pointer to a struct dk_ioctl_data. The i_addr field points to a structure like:

```
structure defect_list {
        struct defect_header defhdr;
        struct defect_entry defentry[NENTS];
```

```
};
```

The i_len field is set to the total length of the structure, which must be less than NBPP from <*sys/param.h*>; at most NENTS defects are returned. The actual number of defects can be determined by examining the defhdr.defect_listlen value, which is the number of bytes returned. This must be divided by the size of the applicable data structure for the type requested. The i_page field should be set to the bits identifying the badblock reporting type. These bits request the combination of manufacturer's and grown defects; and one of bytes from index, physical cyl/head/sec, vendor unique. The only combination that works with all currently supported SCSI disks is type cyl/head/sec; and either combined manufacturer's and grown defects, or just manufacturer's defects.

DIOCREADCAPACITY
: The arg is a pointer to an unsigned integer. The value returned is the number of usable sectors on the drive (as read from the drive).

DIOCSCSIINQ
: The arg is a pointer to a char array at least SCSI_INQUIRY_LEN bytes long. The SCSI inquiry data from the device is copied to this buffer. See a SCSI command specification for details on the structure of this buffer.

DIOCSENSE / DIOCSELECT
: The argument is a pointer to a struct dk_ioctl_data. This allows sending SELECT and SENSE commands to the drive. See the ANSI SCSI specification and individual manufacturer's manuals for allowed page numbers and valid values. Only programs running with superuser permissions can use the DIOCSELECT ioctl.

DIOCSTARTSTOP
: This command issues a SCSI STARTSTOP command to the opened device. The first two bits of the arg are or'd into byte 4 of the SCSI command. See a SCSI command specification for details on this command.

DIOCTEST
: issues the SCSI "Send Diagnostic" command to the drive. The exact tests done are manufacturer specific. The *ioctl* call returns 0 upon success, or sets *errno* to EIO and returns –1 upon failure.

FILES

/dev/dsk/dks*
/dev/rdsk/dks*
/dev/dsk/jag*
/dev/rdsk/jag*
/dev/root
/dev/usr
/dev/swap

SEE ALSO

dvhtool(1M), fx(1M), prtvtoc(1M).

NOTE

The driver attempts to negotiate synchronous SCSI mode with the drive when it is opened. When supported by the drive, this results in greater disk throughput, and better SCSI bus utilization when multiple devices are attached to the SCSI bus. If problems occur because of this (due to use of unsupported drives that don't properly handle this negotiation), you can disable this negotiation by changing the *wd93_syncenable* and *wd93_syncperiod* variables in the file */var/sysgen/master.d/wd93* as described by the comments in that file, and linking a new kernel with *lboot*(1M).

Another variable of possible interest in */var/sysgen/master.d/wd93* is *wd93_enable_disconnect*. If this variable is set to 0, the SCSI host adapter is programmed to NOT support device disconnects. This can be useful with devices that don't support disconnect, which might otherwise cause SCSI bus timeouts. It is also useful with devices that purport to support disconnect, but which don't work correctly when it is enabled. The capability to configure these features is not currently supported for the jag controllers. See the *wd95_bus_flags* array in */var/sysgen/master.d/wd95* for similar capabilities on the wd95 scsi bus.

The kernel automatically disables the synchronous SCSI mode for systems that do not support this feature regardless of what is set in *wd93_syncenable*.

NAME

intro – introduction to special files

DESCRIPTION

This section describes various special files that refer to specific hardware peripherals, and IRIX 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 IRIX system device driver are discussed where applicable.

Disk device filenames are described in *dks*(7M).

Tape device filenames are in the following format:

/dev/rmt/*typecontroller***d***unit*{**nr**}{**ns**|**s**}{**v**}{**stat**}{**.***density*}

Where:

type Identifies the controller type.

controller Indicates the controller number.

unit Indicates the device attached to the controller.

nr Indicates a non rewinding interface.

ns Indicates bytes are not swapped. This should be used for all tape types other than QIC, in almost all cases. It should also be used for QIC tapes imported from or exported to systems other than IRIX.

s Indicates bytes are swapped by the driver, primarily for backwards compatibility with older Silicon Graphics systems.

v Indicates that the variable blocksize device should be used. This device writes a single logical block per read or write system call. The fixed block device transfers 1 or more logical blocks per read or write system call. Variable mode is preferred on 9-track, DAT, and often when importing or exporting 8mm media from non-IRIX systems.

stat A special device that can be used only for the **MTIOCGET** ioctl. All other I/O requests and ioctls fail with the **EINVAL** errno.

.density Optionally specifies the media density, where appropriate. For devices with only one density setting, *density* can be omitted. The **.** is used to keep the *unit* from visually merging with the *density*.

The */dev/mt* directory exists as a link to */dev/rmt* as a portability aid; IRIX does not support block mode tape access. Not all tape types support all of these options. For backwards compatibility, a tape device with neither **ns**, nor **s** is created. It is normally the same device as the **s** device for QIC tapes and **ns** for all other tape types.

SEE ALSO

MAKEDEV(1M), fx(1M), hinv(1M), mt(1), prtvtoc(1M).

NOTE

The other devices in section 7 can be listed with:

```
man -w 7 '*'
```

or all of them can be read with:

```
man 7 '*'
```

NAME

keyboard – keyboard specifications

DESCRIPTION

The keyboard is an up-down encoded 101-key keyboard.

The keyboard connects to the main electronics cabinet through a shielded partially coiled cord and is detachable at the system cabinet only. The mouse plugs into either side of the keyboard. Ports are provided on both sides of the enclosure to allow access to left-handed and right-handed mouse connectors. The keyboard cord contains low voltage direct current power feeds and two serial links; one for the mouse and one for the keyboard. The keyboard serial link is bidirectional, allowing for control of indicator lights and other keyboard functions. Each time a key is pressed or released, a code is sent via the keyboard serial link. Every key has a different upcode and downcode. All keys function the same way, allowing the system software to use keys in any manner. Auto-repeat is the only function that treats keys differently. When auto-repeat is enabled, a subset of the keys repeat when held down. Multiple key presses/releases result in all key transitions being reported.

Electrical Interface

The keyboard serial I/O interface uses RS423 levels and communicates asynchronously to the system at 600 baud. The format used is one start bit followed by eight data bits, an odd parity bit and one stop bit, with one byte sent per key up or down transition. The idle state and true data bits for the interface are Mark level or -5V, whereas false data bits and the start bit are spaces or +5V.

The pin assignments for the keyboard connector on machines with DB-15 connectors are shown in the following table:

KEYBOARD CONNECTOR PINOUT

Pin	Signal	Description
1	GND	Ground
2	GND	Ground
3	GND	Ground
4	KTXD	Keyboard Transmit
5	KRCD	Keyboard Receive
7	+12Vdc	Power
8	+12Vdc	Power
9	+12Vdc	Power
10	MTXD	Mouse Transmit
11	NC	Reserved
12	NC	Reserved
15	-12Vdc	Power

The pin assignments for the DB-9 keyboard connector on the Personal IRIS 4D/20 and 4D/25 machines are shown in the following table for both the CPU connector and the connectors on the keyboard itself. The connectors on each side of the keyboard are identical, so the mouse can be attached on either side.

KEYBOARD CABLE PINOUT

Pin	CPU Signal	Keyboard Signal	Description
1	NC	+5Vdc	Power
2	KRCD	KTXD	Keyboard to CPU
3	NC	-5Vdc	Power
4	-12Vdc	-12Vdc	Power
5	MRCD	MTXD	Mouse to CPU
6	GND	GND	Ground
7	+12Vdc	+12Vdc	Power
8	KTXD	KRCD	CPU to Keyboard
9	GND	GND	Ground

The pin assignments for the DIN-6 keyboard connector on the CPU board of some of the newer systems are shown in the following table:

KEYBOARD CABLE PINOUT

Pin	Signal	Description
1	KRCD	Keyboard Receive
2	MRCD	Mouse Receive
3	GND	Ground
4	+8Vdc	Power
5	KTXD	Keyboard Transmit
6	-8Vdc	Power

The pin assignments for the mouse port connector (on the keyboard, not on the CPU) for keyboards with DB9 connector are shown in the following table; either connector on the keyboard can be used for the mouse or for the keyboard cable.

MOUSE PORT

Pin	Signal	Description
1	+5V	Power
3	-5V	Power
5	MTXD	Transmit Data
9	GND	Ground

For machines whose keyboards have the DIN-6 connector on the keyboard, as well as on the CPU, the pinout is shown in the following table. The connectors on both sides of the keyboard have identical pinout, either can be used for the mouse and the cable to the CPU.

KEYBOARD CONNECTOR PINOUT

Pin	Signal	Description

1	KTXD	Keyboard Transmit
2	MTXD	Mouse Transmit
3	GND	Ground
4	+8Vdc	Power
5	KRCD	Keyboard Receive
6	NC	Not Connected

Software Interface

The interface between the keyboard and the system is 600 baud asynchronous. The format used is one start bit followed by eight data bits, an odd parity bit and one stop bit, with one byte sent per key up or down transition. The MSB of the byte is a **0** for a downstroke and a **1** for an upstroke. Control bytes are sent to the keyboard with the same speed and format. The system software does all the processing needed to support functions such as capitalization, control characters, and numeric lock. Auto-repeat for a specified set of characters can be turned on or off by the system software by sending a control byte to the keyboard. When auto-repeat is enabled, a pressed key begins auto-repeating after 0.65 seconds and repeats 28 times per second. The keyboard initializes upon power-up. The configuration request control byte causes the keyboard to send a two-byte sequence to the system. The second byte contains the eight-bit value set on a DIP switch in the keyboard. All keyboard lights (if any; some newer systems have keyboards without user controllable lights) are controlled by the system software by sending control bytes to the keyboard to turn them on or off. Control bytes are also used for long and short beep control and key click disable. When key click is enabled, the keys click when they are pressed. The long beep duration is 1 second and the short beep duration is 0.2 second. There are three lights labeled NUM LOCK, CAPS LOCK, and SCROLL LOCK that are under software control. On older keyboards there are also four general-purpose keyboard lights labeled L1 through L4. The required keycode mappings and control byte formats are shown in the following tables. Note that the legend names prefixed by two asterisks are reserved and not implemented on the keyboard. Legend names prefixed by two exclamation marks do NOT have the auto-repeat enable capability. Legend names prefixed by two dollar signs do NOT have the key click enable capability.

LEGENDS VS KEYCODES IN DECIMAL

Legend	Code
AKEY	10
BKEY	35
CKEY	27
DKEY	17
EKEY	16
FKEY	18
GKEY	25
HKEY	26
IKEY	39
JKEY	33
KKEY	34

LKEY	41
MKEY	43
NKEY	36
OKEY	40
PKEY	47
QKEY	9
RKEY	23
SKEY	11
TKEY	24
UKEY	32
VKEY	28
WKEY	15
XKEY	20
YKEY	31
ZKEY	19
ZEROKEY	45
ONEKEY	7
TWOKEY	13
THREEKEY	14
FOURKEY	21
FIVEKEY	22
SIXKEY	29
SEVENKEY	30
EIGHTKEY	37
NINEKEY	38

LEGENDS VS KEYCODES IN DECIMAL

Legend	Code
**!!BREAKKEY	0
**!!SETUPKEY	1
$$!!LEFTCTRL	2
$$!!CAPSLOCKKEY	3
$$!!RIGHTSHIFTKEY	4
$$!!LEFTSHIFTKEY	5
**!!NOSCRLKEY	12
!!ESCKEY	6
!!TABKEY	8
RETURN.ENTER	50
SPACEKEY	82
**LINEFEEDKEY	59
BACKSPACEKEY	60

DELKEY	61
SEMICOLONKEY	42
PERIODKEY	51
COMMAKEY	44
QUOTEKEY"	49
ACCENTGRAVEKEY~	54
MINUSKEY	46
VIRGULEKEY?	52
BACKSLASHKEY	56
EQUALKEY	53
LEFTBRACKETKEY	48
RIGHTBRACKETKEY	55
LEFTARROWKEY	72
DOWNARROWKEY	73
RIGHTARROWKEY	79
UPARROWKEY	80
PAD0	58
PAD1	57
PAD2	63
PAD3	64
PAD4	62
PAD5	68
PAD6	69

LEGENDS VS KEYCODES IN DECIMAL

Legend	Code
PAD7	66
PAD8	67
PAD9	74
**PADPF1	71
**PADPF2	70
**PADPF3	78
**PADPF4	77
PADPERIOD	65
PADMINUS	75
**PADCOMMA	76
!!PADENTER	81
$$!!LEFTALT	83
$$!!RIGHTALT	84
$$!!RIGHTCTRL	85
F1	86

F2	87
F3	88
F4	89
F5	90
F6	91
F7	92
F8	93
F9	94
F10	95
F11	96
F12	97
!!PRINT.SCREEN	98
$$!!SCROLL.LOCK	99
!!PAUSE	100
!!INSERT	101
!!HOME	102
!!PAGEUP	103
!!END	104
!!PAGEDOWN	105
$$!!NUM.LOCK	106
PAD.BKSLASH/	107

LEGENDS VS KEYCODES IN DECIMAL

Legend	Code
PAD.ASTER*	108
PAD.PLUS+	109
config byte(1st of 2 bytes)	110
config byte(2nd of 2 bytes)	DIP SW
GERlessThan	111
spare1	112
spare2	113
spare3	114
spare4	115
spare6	117
spare7	118
spare8	119
spare9	120
spare10	121

KEYCODES IN DECIMAL VS LEGENDS

Code	Legend

0	**BREAKKEY
1	**!!SETUPKEY
2	$$!!LEFTCTRL
3	$$!!CAPSLOCKKEY
4	$$!!RIGHTSHIFTKEY
5	$$!!LEFTSHIFTKEY
6	!!ESCKEY
7	ONEKEY
8	!!TABKEY
9	QKEY
10	AKEY
11	SKEY
12	**!!NOSCRLKEY
13	TWOKEY
14	THREEKEY
15	WKEY
16	EKEY
17	DKEY
18	FKEY
19	ZKEY
20	XKEY
21	FOURKEY
22	FIVEKEY
23	RKEY
24	TKEY
25	GKEY
26	HKEY
27	CKEY
28	VKEY
29	SIXKEY
30	SEVENKEY
31	YKEY
32	UKEY
33	JKEY
34	KKEY
35	BKEY

KEYCODES IN DECIMAL VS LEGENDS

Code	Legend
36	NKEY
37	EIGHTKEY

38	NINEKEY
39	IKEY
40	OKEY
41	LKEY
42	SEMICOLONKEY
43	MKEY
44	COMMAKEY
45	ZEROKEY
46	MINUSKEY
47	PKEY
48	LEFTBRACKET
49	QUOTEKEY
50	RETURN.ENTER
51	PERIODKEY
52	VIRGULEKEY
53	EQUALKEY
54	ACCENTGRAVEKEY
55	RIGHTBRACKETKEY
56	BACKSLASHKEY
57	PADONEKEY
58	PADZEROKEY
59	**LINEFEEDKEY
60	BACKSPACEKEY
61	DELETEKEY
62	PADFOURKEY
63	PADTWOKEY
64	PADTHREEKEY
65	PADPERIODKEY
66	PADSEVENKEY
67	PADEIGHTKEY
68	PADFIVEKEY
69	PADSIXKEY
70	**PADPF2KEY
71	**PADPF1KEY

KEYCODES IN DECIMAL VS LEGENDS

Code	Legend
72	LEFTARROWKEY
73	DOWNARROWKEY
74	PADNINEKEY
75	PADMINUSKEY

76	**PADCOMMAKEY
77	**PADPF4KEY
78	**PADPF3KEY
79	RIGHTARROWKEY
80	UPARROWKEY
81	!!PADENTERKEY
82	SPACEKEY
83	$$!!LEFTALT
84	$$!!RIGHTALT
85	$$!!RIGHTCTRL
86	F1
87	F2
88	F3
89	F4
90	F5
91	F6
92	F7
93	F8
94	F9
95	F10
96	F11
97	F12
98	!!PRINT.SCREEN
99	$$!!SCROLL.LOCK
100	!!PAUSE
101	!!INSERT
102	!!HOME
103	!!PAGEUP
104	!!END
105	!!PAGEDOWN
106	$$!!NUM.LOCK
107	PAD.BKSLASH/

KEYCODES IN DECIMAL VS LEGENDS

Code	Legend
108	PAD.ASTER*
109	PAD.PLUS+
110	config byte(1st of 2 bytes)
DIP SW	config byte(2nd of 2 bytes)

CONTROL BYTES RECOGNIZED BY KEYBOARD

BIT TRUE	DESCRIPTION	
	BIT 0 = 0	BIT 0 = 1
1	short beep	complement ds1 and ds2
2	long beep	ds3
3	click disable	ds4
4	return configuration byte	ds5
5	ds1	ds6
6	ds2	ds7
7	enable auto-repeat	not used

DISPLAY LABELS

DISPLAY DESIGNATION	LABEL
ds1	NUM LOCK
ds2	CAPS LOCK
ds3	SCROLL LOCK
ds4	L1
ds5	L2
ds6	L3
ds7	L4

NOTE

Indy and Indigo^2 use a PS/2 style keyboard (detailed in *pckeyboard*(7)) that uses a different scan code set. This difference may break compatibility for some programs that operate with raw scan codes.

SEE ALSO

keyboard(1), mouse(7), pckeyboard(7), pcmouse(7).

NAME

mouse – mouse specifications

DESCRIPTION

Signals

The serial data interface signal level is compatible with RS-423, which has roughly a 10V swing centered about ground. The idle state and true data bits for the interface are Mark level or –5V whereas false data bits and the start bit are spaces or +5V. The serial data is transmitted at 4800 baud with one start bit, eight data bits, and no parity.

Protocol

The mouse provides a five-byte data block whenever there is a change of position or button state. The first byte is a sync byte that has its upper five bits set to 10000 and its lower three bits indicating the button states where a 0 indicates depression. The sync byte looks like this: 10000LMR. The next four bytes contain two difference updates of the mouse's change in position: X1, Y1, X2, and Y2. Positive values indicate movement to the right or upward. System software ignores bytes beyond the first five until reception of the next sync byte.

Pinout

The pinout of the mouse connector on the keyboard is on the *keyboard*(7) reference page.

NOTE

Indy and Indigo^2 use a PS/2 style mouse, which is described in *pcmouse*(7).

SEE ALSO

keyboard(7), pckeyboard(7), pcmouse(7).

NAME

mtio – magnetic tape interface

DESCRIPTION

mtio describes the generic interface provided for dealing with the various types of magnetic tape drives supported on Silicon Graphics systems. 1/4" (QIC) cartridge tapes, 1/2" nine-track tapes, 8 mm video tapes, NTP, STK 9490, STK SD3, DAT (digital audio tapes), and DLT are currently supported. (Not all systems support all tapedrives.)

Tape drives are accessed through special device files in the */dev/mt/** and */dev/rmt/** directories. The *mt* directory is a link to the *rmt* directory for ease of porting; the block interface sometimes associated with the devices in *mt* is not supported under IRIX. Refer to *intro*(7) for a general description of the naming conventions for device files in theses subdirectories. Naming conventions for the specific devices are listed under *tps*(7M).

Normally the device specific name is linked to a user friendly synonym for ease of use. Many commands that manipulate magnetic tape refer to these synonyms rather than device specific names. There are up to four user-friendly device synonyms:

/dev/tape This is the tape unit that is the default system tape drive. It is linked to one of the specific device names in */dev/rmt*. This device rewinds the tape when closed. For QIC tapes, the devices linked to **/dev/tape** do software byte swapping to be compatible with the IRIS 2000 and 3000 series systems; the non-byte swapping device is also available, and should almost always be used for all tape types other than QIC.

/dev/nrtape Same as **/dev/tape**, except the tape is not rewound when closed.

/dev/tapens Same as **/dev/tape**, except no byte swapping is done; normally created only for QIC tapes.

/dev/nrtapens Same as **/dev/nrtape**, except no byte swapping is done; normally created only for QIC tapes.

See the (unfortunately somewhat confusing) script */dev/MAKEDEV* for details of which devices are linked to **/dev/tape** for each tape type. In particular, look at the **tapelinks** target for the default links. Also be aware that if a **/dev/tape** exists as a link to a valid tape device, it is left as is, in order to preserve local changes.

Note that even the norewind tape devices can be rewound by the system, in some cases. In particular, all tapes are rewound on first use after a system boot, and when detected by the driver, are rewound after the tape has been changed. DAT drives are also rewound when the drive is switched between audio and data modes. This means that if you wish to append a new dataset to a tape that already contains datasets, you should always issue the **mt feom** command AFTER loading the tape, just prior to using the program that will append to the tape. Do not count on a tape remaining at EOD, just because that is where it was before it was removed.

The system makes it possible to treat the tape similar to any other file, with some restrictions on data alignment and request sizes. Seeks do not have their usual meaning (that is, they are ignored) and it is not possible to read or write other than a multiple of the fixed block size when using the fixed block device. Writing in very small blocks (less than 4096 bytes) is inadvisable because this tends to consume more tape (create large record gaps verses data for 9-track and may pad to a device specific boundary for others, such as 8mm, if the drive isn't kept streaming); it may also cause the tape to stop its streaming motion, increasing wear on the drive and decreasing throughput.

The standard QIC tape consists of a series of 512-byte records terminated by an end-of-file. Other tape devices (such as 9-track, 8 mm, and DAT) typically support both fixed size block format and variable size blocks format.

When using the variable format, there is an upper limit to the size of a single read or write, typically the size of the RAM buffer on the drive. At this time, the upper limit is 64K bytes on 9-track and 240K bytes on the 8 mm drives. This information can be obtained by use of the **MTIOCGETBLKINFO** ioctl (for SCSI tape drives only). The main use of this format is to allow small header blocks at the beginning of a tape file, while the rest are typically the same (larger) size.

When the fixed block size device is used, the size of a single read or write request is limited only by physical memory. Currently the default size is 1024 bytes on 8mm, and 512 bytes for all others. This size can be reset with the **MTSCSI_SETFIXED** ioctl or the **mt setblksz XXX** command; the value remains set until the next boot or reset via ioctl. If the variable blocksize device is used, the block size reverts to the default on the next use of the fixed blocksize device. The default fixed blocksize for the *tps* driver is set from the table in the */var/sysgen/master.d/scsi* file, as are the drive types, based on the inquiry data returned by the drive.

A tape is normally open for reading and/or writing, but a tape cannot be read and written simultaneously. After a rewind, a space, an unload, or an **MTAFILE** ioctl, writes can follow reads and vice-versa, otherwise only reads, or only writes can be done unless the tape is first closed; performing an **MTWEOF** ioctl is considered to be a write operation in this sense; after reading, an **MTWEOF** is not allowed unless one of the operations above is first executed, or the tapedrive is closed and reopened.

Whenever the tape is closed after being written to, a file-mark is written (2 on 9-track tapes) unless the tape has been unloaded or rewound just prior to the close; if the last operation before the close is an **MTWEOF**, no additional filemarks are written at close.

Some tape drives support more than one speed; for SCSI 9-track tape drives, the default is 100 ips (streaming mode); this can be set to 50 ips by using the **MTSCSI_SPEED** ioctl. Some tape drives such as the Kennedy 96XX models do auto density select when reading; this can be disabled only by using the drive's front panel setup mode.

The **MTANSI** ioctl allows writing of ANSI 3.27 style labels after the early warning point (naturally, drives that don't support variable sized blocks won't support 80 byte labels). It is currently implemented only for SCSI tape drives. It remains set until the next close, or reset with a 0 argument. If used, standard Silicon Graphics EOT handling for *tar*, *bru*, and *cpio* won't work correctly while set. An arg of 1 enables, 0 disables. NOTE: When the EOT marker is encountered, the current I/O operation returns either a short count (if any I/O completed), or –1 (if no I/O completed); it is the programmers responsibility to determine if EOT has been encountered by using **MTIOCGET** or logic internal to the program. This ioctl should be issued AFTER encountering EOT, if ANSI 3.27 EOT handling is desired. Subsequent I/Os are then allowed and return the count actually transferred or –1 if the drive was unable to transfer any data. The standard calls for writing a FM before the label. If this is not done, the drive may return label info as data.

IOCTL OPERATIONS

Different drivers and drives support different tape commands, in the form of ioctl's. These operations and their structures are defined in the file */usr/include/sys/mtio.h*, which has fairly extensive comments. All drivers support some common definitions of tape status via the **MTIOCGET** ioctl; in particular, the bits defined for the *mt_dposn* field of the *mtget* structure are the same for all the IRIX devices. other fields are driver specific, and the appropriate header files should be consulted for the meaning of these fields.

Those ioctls that are not supported for a particular drive or driver normally return EINVAL. For example, the **MTAFILE** ioctl returns EINVAL if the tape drive is not a device that supports overwrite (currently only 9-track and DAT), since QIC and 8mm drives only allow appending at EOD and, for 8mm, from the BOT side of a FM.

NOTES

When a tape device is opened, the tape is rewound if there has been a media change, or the drive has gone offline, or there has been a bus reset (normally only after a reboot or powerup); otherwise the tape remains at the same position as at the previous close. See the specific driver manual pages for more details.

EXAMPLE

The following code fragment opens the default no rewind tape device, positions it to the 2nd filemark, and then rewinds it. This is typical of the use of most of the op codes for the **MTIOCTOP** and **ABI_MTIOCTOP** ioctls; the latter is for use by programs compliant with the MIPS ABI and, other than the name, functions identically to the former.

```
#include <sys/types.h>
#include <sys/mtio.h>
main()
{
     struct mtop op;
     int fd;

     if((fd = open("/dev/nrtape", 0)) == -1)
          perror("can't open /dev/tape");
```

```
        op.mt_op = MTFSF;
        op.mt_count = 2; /* number of fmk's to skip */
        if(ioctl(fd, MTIOCTOP, &op) == -1)
             perror("ioctl to skip 2 FMs fails");

        op.mt_op = MTREW; /* mt_count field is ignored for MTREW */
        if(ioctl(fd, MTIOCTOP, &op) == -1)
             perror("ioctl to rewind fails");
    }
```

FILES

/dev/tape
/dev/nrtape
/dev/tapens
/dev/nrtapens
/dev/rmt/*

SEE ALSO

MAKEDEV(1M), bru(1), cpio(1), mt(1), tar(1), dataudio(3), rmtops(3), datframe(4), tps(7M).

NAME

pcmouse – mouse specifications

DESCRIPTION

Indy and Indigo2 systems uses an industry standard PC (6-pin mini-DIN) mouse port. This is a departure from other Silicon Graphics systems and allows easy attachment of third party mice, trackballs, or other pointing devices.

Compatibility

Third-party mouse-port-compatible (also called "PS/2 compatible," "Pointing Device Port" or "PDP") pointing devices commonly sold into the IBM PC-compatible market can be used. Note that the common Microsoft-compatible serial mice do not work on the mouse port. Three-button devices are preferred, but two-button devices operate as if the middle button does not exist.

The following mice have been tested (many only briefly):

SGI Indigo^2 mouse
SGI Indy mouse
IBM PS/2 Model 6450350 (old style)
IBM PS/2 Model 33G5430 (new style)
IBM RS/6000 P/N 11F8895 (Logitech M-SB9-6MD)
Alps Glidepoint (PS/2 mouse port version)
Logitech MouseMan Serial & Mouseport Version
Logitech TrackMan Serial & Mouseport Version
Logitech M-SE9-6MD
Logitech M-SF15-6MD
Microsoft Mouse Serial & PS/2 Version (only two buttons)
MicroSpeed MicroTRAC trackball (only two buttons)
Mouse Systems White Mouse
Mouse Systems PC Mouse III (in two button mode only)

Protocol

The PS/2 mouse uses a three-byte data block. The first byte is a control byte with the format: YO XO YS XS F M R L where [XY]O is an overflow indicator, [XY]S is the sign of the delta bytes, F is floats depending on the particular mouse used, and M R L are middle, right, and left buttons (1 indicates pressed) respectively. Byte two is the X delta and byte three is the Y delta.

Pinout

The mouse connector is a 6 pin mini-DIN connector with the shield connected to the system chassis:

PIN ASSIGNMENTS

Pin	Description
1	Data
2	Reserved
3	Signal Ground
4	Power +5V
5	Clock
6	Reserved

NOTES

All Silicon Graphics systems except Indy and Indigo^2 use the mouse described in *mouse*(7).

Some Silicon Graphics mice (particularly Silicon Graphics PN 9150800) look very similar to the Indigo^2 mouse (Silicon Graphics PN 9150809) and even use the same 6-pin mini-DIN connector, but are not compatible.

SEE ALSO

prom(1M), keyboard(7), mouse(7), pckeyboard(7).

NAME

root, rroot, usr, rusr, swap, rswap – partition names

DESCRIPTION

root, **rroot**, **usr**, **rusr**, **swap**, and **rswap** are the device special files providing access to important partitions on the root disk drive of a system. These links are made by the *MAKEDEV*(1m) script, and map to fixed partitions, even if not used that way by local conventions. Therefore it is best not to change these links, if you intend to use different partition layouts, but rather to use the full device name (*/dev/dsk/dks**) instead, particular in *fstab*(4). The names beginning with **r** are the raw (character) device access; the others are the block device access, which uses the kernel buffer system.

The standard system drive partition allocation shipped by Silicon Graphics has **root** on partition 0 and **swap** on partition 1. Partition 7 is the entire usable portion of the disk (excluding the volume header) and is normally used for *option* drives, rather than the system drive. Partition 8 is the volume header (see *vh*(7M), *prtvtoc*(1M), and *dvhtool*(1M)). Partition 10 (**vol**) is the entire drive.

The standard system with SCSI drives usually has */dev/root* linked to */dev/dsk/dks0d1s0*, */dev/swap* linked to */dev/dsk/dks0d1s1*, and (if / and /usr are seperate filesystems, a *usrroot* partitioning), */dev/usr* linked to */dev/dsk/dks0d1s6*.

FILES

/dev/dsk/dks*
/dev/rdsk/dks*
/dev/root
/dev/usr
/dev/swap
/dev/rvh

SEE ALSO

MAKEDEV(1M), dvhtool(1M), fx(1M), prtvtoc(1M), fstab(4), dksc(7M), vh(7M).

NAME

serial – serial communication ports

SYNOPSIS

/dev/tty[**dmf**][**1-56**]

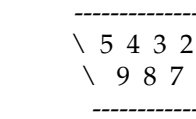
DESCRIPTION

All Silicon Graphics systems have two or more general purpose serial ports. These ports can be used to connect terminals, printers, modems, other systems, or graphical input devices such as a tablet or dial and button box. Each line can be independently set to run at any of several speeds, as high as 19,200 or even 38,400 bps. Various character echoing and interpreting parameters can also be set. See *stty*(1) and *termio*(7) for details on the various modes.

Details of the serial ports found on optional add-on boards are given elsewhere. The Audio/Serial Option for CHALLENGE/Onyx provides six high-speed serial ports, see *asoser*(7) for more information. The CDSIO VME board provides six serial ports; see *cdsio*(7) for more information.

Special files for the serial ports exist in the */dev* directory. These files, **tty**[**dfm**][**1-56**], are created automatically by *MAKEDEV*(1M) when system software is installed. Each port is referred to by three different names, **/dev/ttyd***nn*, **/dev/ttym***nn*, and **/dev/ttyf***nn*, where *nn* represents the port number. Opening the **ttyd**, **ttym**, or **ttyf** versions of a port enables different signals and modes on the communication line. Typically, the **ttyd** version is used for directly connecting simple devices including most terminals; **ttym** is used for devices that use modem control signals; and **ttyf** is used for devices that understand hardware flow control signals.

There are four different types of connectors found on various 4D models. The DB-9 serial port connectors, which are found on the CHALLENGE, Onyx, Personal IRIS, and POWER series systems, have the following pin assignments.

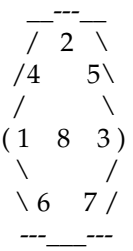


Pin	Name	Description
2	TD	Transmit Data
3	RD	Receive Data
4	RTS	Request To Send
5	CTS	Clear To Send
7	SG	Signal Ground
8	DCD	Data Carrier Detect
9	DTR	Data Terminal Ready

The CHALLENGE and Onyx systems provide an RS-422 port. This RS-422 port uses a DB-9 serial connector and has the following pin assignments.

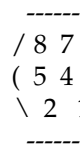
Pin	Name	Description
1	DTR	Data Terminal Ready
2	TxD-	Transmit Data -
3	RxD-	Receive Data -
4	DCD	Data Carrier Detect
5	CTS	Clear To Send
6	SG	Signal Ground
7	TxD+	Transmit Data +
8	RxD+	Receive Data +
9	RTS	Request to send

In order to support peripherals that draw power from the host system, the CHALLENGE and Onyx systems provide two powered-peripheral serial ports. These ports have a DIN-8 connector. The powered ports share the tty2 and tty3 signal lines with the standard DB-9 connectors; if the DB-9 connector for tty2 is already in use, you cannot use the powered peripheral connector for tty2. Similarly, if tty3's DB-9 connector is connected to a peripheral, the powered peripheral port connected to the tty3 signal lines cannot also be used. The powered peripheral ports have the following pin assignments.



Pin	Name	Description
1	DTR	Data Terminal Ready
2	CTS	Clear To Send
3	STEREO	Stereo field sync
4	RD	Receive Data
5	TD	Transmit Data
6	SG	Signal Ground
7	GND	Ground point
8	V10P	10V supply

The DIN-8 serial port connectors on the Personal IRIS 4D/30, 4D/35, 4D/RPC (Indigo), 4D/RPC-50 (R4000 Indigo), Indy, and Indigo^2 have the following pin assignments.



4D Compatible Pin Assignments (RS-232)

Pin	Name	Description
1	DTR	Data Terminal Ready
2	CTS	Clear To Send
3	TD	Transmit Data
4	SG	Signal Ground
5	RD	Receive Data
6	RTS	Request To Send
7	DCD	Data Carrier Detect
8	SG	Signal Ground

Macintosh SE Compatible Pin Assignments (RS-422)

Pin	Name	Description
1	HSKo	Output Handshake
2	HSKi	Input Handshake Or External Clock
3	TxD-	Transmit Data -
4	GND	Signal Ground
5	RxD-	Receive Data -
6	TxD+	Transmit Data +
7	GPi	General Purpose Input
8	RxD+	Receive Data +

The set of signals that are actually used depends upon which form of the device was opened. If the **ttyd** name was used, only **TD**, **RD**, and **SG** signals are meaningful. These three signals are typically used with "dumb" devices that either do not need any sort of data flow control or use software flow control (see the description of the **ixon**, **ixany**, and **ixoff** options in *stty*(1) for more information on setting up software flow control). If the **ttym** device is used, the **DCD**, and **DTR** signals are also used. These signals provide a two way handshake for establishing and breaking a communication link with another device and are normally used when connecting via a modem. When the port is initially opened, the host asserts the **DTR** line and waits for the **DCD** line to become active. If the port is opened with the **O_NDELAY** flag, the open succeeds even if the **DCD** line is not active. A hangup condition occurs if the **DCD** line transitions from active to inactive. See *open*(2), and *termio*(7) for more information. If the **ttyf** device is used, all of the signals are used. The additional signals provide for full hardware flow control between the host and the remote device. The **RTS** line is asserted by the host whenever it is capable of receiving more data. The

CTS line is sampled before data is transmitted and if it is not active, the host suspends output until it is.

The DIN-8 serial port connectors on the Personal IRIS 4D/30, 4D/35, 4D/RPC, 4D/RPC-50, Indy, and Indigo^2 can be used to communicate with serial devices using RS-422 protocol. User can use the stream ioctl commands, **SIOC_EXTCLK** and **SIOC_RS422**, defined in */usr/include/sys/z8530.h* to switch between internal/external clock and RS-232/RS-422 protocols. Another command that can be useful is **SIOC_ITIMER**; it informs the driver how long it should buffer up input data, in clock ticks, before sending them upstream. Data can sometimes be sent upstream before, but never after, this time limit. This feature reduces the cpu cost of receiving large amounts of data by sending data upstream in large chunks. This duration can also be configured into the kernel by tuning the duart_rsrv_duration variable.

CHALLENGE AND ONYX L/XL PORT CONFIGURATION

By default, Onyx and CHALLENGE L/XL systems enable only the serial ports of the master IO4. To enable the serial ports on other IO4 boards, a vector line must be added for the epcserial device to */var/sysgen/system/irix.sm*. The following vector line configures the serial ports on the IO4 in slot 13 as tty45, tty46, and tty47:

```
VECTOR: bustype=EPC module=epcserial unit=1 slot=13
```

The first two options (bustype and module) are mandatory and tell *lboot*(1M) that you're configuring serial ports. The **unit** option specifies which set of tty names should be used for this set of ports: unit 1 corresponds to the logical devices tty45, tty46, and tty47; unit 2 represents devices tty49, tty50, and tty51; unit 3 specifies devices tty53, tty54, and tty55. Finally, the **slot** option indicates which IO4 board contains the ports that should be mapped. Each board must have its own vector line. Configuring one IO4 board's serial ports has no effect on any of the other boards. After **irix.sm** has been updated, the *autoconfig*(1M) command should be issued to reconfigure the kernel. It may also be necessary to execute *MAKEDEV*(1M) in order to build device files for the new ports.

If the system is unable to honor the VECTOR line for some reason (if, for example, the specified slot is invalid), a warning message is written to */var/adm/SYSLOG*. These warning messages contain the string **epcserial**, in order to facilitate finding them with commands like *grep*(1). Because the console port has not been initialized when these messages are issued, the kernel is unable to display the warning on the console.

Only the master IO4 provides an RS-422 port (tty4). Additional IO4 boards support three RS-232 serial ports only. To allow for future expansion, however, space was left in the serial port namespace for the additional RS-422 ports. For this reason, there is no actual device associated with tty48, tty52, and tty56.

FILES

/dev/tty[dmf][1-4,45-56]
/usr/include/sys/z8530.h
/dev/MAKEDEV
/var/sysgen/system

SEE ALSO

system(4), asoser(7), cdsio(7), keyboard(7), streamio(7), termio(7).

NAME

tps, tpsc, jagtape – SCSI tape interface

SYNOPSIS

/dev/rmt/tps*
/dev/rmt/jag*

DESCRIPTION

Silicon Graphics systems support the Small Computer System Interface (SCSI) for various tape drives, including QIC24 and QIC150 1/4" cartridges, 9-track, 8 mm video, DLT (digital linear tape), NTP, STK 9490, STK SD3, and DAT (digital audio tape) tape drives. Not all systems support all tape drives. Since so many different types of devices are supported, and not all their features can be determined directly from the drive, a configuration table defines their capabilities. This is usually found in the file */var/sysgen/master.d/scsi*.

The special files are named according to this convention:

/dev/{**r**}**mt/tps***controller***d***ID*{**nr**}{**ns**}{**s**}{**v**}{**.***density*}{**c**}
/dev/rmt/jag*controller***d***ID*{**nr**}{**ns**}{**s**}{**v**}{**stat**}{**.***density*}

controller is the SCSI controller number and *ID* is variously known as the SCSI ID, the SCSI address, the drive address, and the unit number. The device types are:

{**nr**} no-rewind on close device

{**ns**} non-byte swapping device

{**s**} byte swapping device

{**v**} variable block size device (supported only for 9-track, DAT, DLT and 8 mm devices as shipped)

{**stat**} a special purpose device; it can be used even when one of the other names for the same physical device is already opened (see below)

{**.***density*} for 9-track tape it is one of 800, 1600, 3200, or 6250 and for the Exabyte 8500 it is one of 8200 and 8500

{**c**} data compression; it is only supported on DLT and should not be confused with density

These special devices are accessible by only one program at a time, except for the {**stat**} device. Opens on the {**stat**} device can block for several seconds, if another name for the same devices is being opened or closed at the same time. Similarly, the MTIOCGET ioctl on the {**stat**} device can block if long operations are in progress via one of the other names for the same device.

The only operation the {**stat**} device supports is the MTIOCGET ioctl, open, and close; all other attempted operations cause the EINVAL errno to be returned. It never causes any tape movement of any kind (in particular, it never tries to load the tape, even if media is present).

Typically, if this tape drive is used as the system tape drive, the device-specific names described above are linked to user-friendly names in the */dev* directory. See **NOTES** below and *mtio*(7) for a description of the user-friendly names.

SPECIAL FEATURES

The different devices support a multitude of capabilities. In particular, some support multiple densities, some support fixed block size only, some support variable block sizes, some support multiple speeds, some support direct transfer of audio data over the SCSI bus. Some of these features are selected by which minor device is opened, and others must be set via ioctl commands. The list of capabilities that a particular drive supports is set by the **MTCAN_*** bits set in the *master.d/scsi* file for that drive. When a capability required for an operation is not set, the EINVAL error code is returned for the request.

In particular, in audio mode the variable block size device must always be used for I/O, and parameters such as the recording frequency, program number, and so on are all part of the data stream. Additionally, when in audio mode the rewind and seek commands return immediately. If the drive is closed in this state, subsequent opens block until they are completed. If the drive remains open, certain commands, such as MTIOCGET and MTIOCAUDPOSN, can be used to determine the current position of the tape during the seek or rewind operation; the third argument is a pointer to a struct mtaudio. The MTAUD tape op is used to enable and disable audio mode. The third argument is 0 to use data mode and 1 to use audio mode.

Many of the features that require ioctls can be set or changed via the *mt*(1) command.

The data structures and values for these ioctls can be found in the include file */usr/include/sys/mtio.h*, which is shipped with all systems. There are extensive comments in this file, and at this time no attempt has been made to document most of them here or in the *mtio*(7) reference page.

However, there are a few surprising return values that are mentioned here. In particular, when using partitioned tapes (see **mt setpart**, or the MTSETPART ioctl), partition 0 is the 'main' partition, which is the final part of the tape, and partition 1 is the partition closest to BOT. Partitioned tapes are intended primarily so that a tape directory can be written at the beginning of the tape, without any worry of overwriting the data portion of the tape. In addition, partitioned tapes indicate BOT (EOT) when at beginning (end) of partition, rather than the 'real' BOT (EOT).

Some devices support overwrite at arbitrary tape positions, while others require that the tape be at BOT or EOD (end of data). Others allow overwrite at any filemark (that is, 8mm from the BOT side of any filemark). Some experimentation may be necessary to decide what your particular drive supports; all drives support at a minimum writing at both BOT and EOD, assuming the media isn't write protected.

TAPE MOVEMENT CRITERIA

The only time the driver ever moves a tape is when told to, with the exceptions listed below. A command that causes tape movement is always issued, assuming that it is a valid command and the media is loaded.

A significant change from IRIX releases prior to 4.0 is that the driver does NOT automatically position to the next filemark on the first command that does I/O to the tape. This means that **mt bsr**, and **mt fsr** now have meaning since the tape is in fact left where it is positioned. It is now the programmer's or user's responsibility to ensure that the tape is in a valid state for I/O. The drive or driver still detects and prevents operations not valid for the current tape position, returning an appropriate error in *errno*.

1. If the first read after an open encounters a filemark before transferring any data and the tape was not known to be at the top of a filemark or BOT, the filemark is skipped and the read retried. Any further errors are reported exactly as they occur. In particular, if two sequential filemarks are found, the tape is positioned between them.

2. If a read command encounters a filemark, that read returns a short count (if any data was read) or 0. If a short count is returned, the next read returns 0, allowing detection of filemarks. An MTFSF ioctl should not be done at this point, as the tape drive itself has already skipped over the filemark. If an MTFSF is done at this point, the count should be decremented by one. The driver attempts to deal with this case, but can not always do so, due to differences in drive firmware.

   The read following the read that returns 0 returns the data in the next tape file, if any. Note that this is a change from earlier IRIX releases, which required an explicit MTFSF or other tape ioctl to move the tape before further reads could be done. The new behavior is consistent with most actual tape drives and tape drivers in most BSD-derived UNIX systems. Similarly, if an MTFSR ioctl encounters a filemark, it stops at the filemark and subsequent reads return data from the next tapefile, if any.

3. If early warning is encountered on a write or write filemark, the driver does not allow further writes or write filemarks, unless the MTANSI ioctl is issued. Any data remaining is flushed to tape, if possible. For the QIC tape drives and the 8mm drives, an error is returned on both read and write unless all the data was transferred when early warning is encountered. This is so that older multi-volume backups continue to work and new ones can be read on older releases. All newer drives (9 track, DAT, and future drive types) return a short count if not all the data could be transferred. Requests other than read or write are sent to the tape drive as usual.

4. On close, if the last tape movement operation was a successful write (not a write filemark), the following happens:

   a) Half-inch tape drives write two filemarks and then backspace one file, so that the tape is positioned between the two filemarks just written. If, however, the tape is not in ansi mode and early warning has been encountered, no write filemark or backspace is done; filemarks are never written in audio mode.

b) Other tape drives write one filemark. No backspace is done.

5. On the first status request or request that does tape motion after a system boot, a SCSI bus reset, or a tape cartridge insertion, the tape is rewound and loaded; this occurs even for the norewind devices. In addition, some drives rewind a tape when it is ejected. This means that if you wish to append a new dataset to a tape that already contains datasets, you should always issue the **mt feom** command AFTER loading the tape, just prior to using the program that will append to the tape. Do not count on a tape remaining at EOD, just because that is where it was before it was removed.

For devices that support it, the prevent media removal command is sent to the drive on open and released on close, so that eject buttons (when present) are disabled.

ERROR RETURNS

The following errors are returned by this driver; other errors can also be returned by higher levels of the operating system.

EAGAIN The drive returned an error indicating it was not ready (tape ejected, drive taken offline, and so on).

EBUSY Returned on opens when the drive has already been opened.

EFAULT A bad address was passed in a call that required a data transfer.

EINVAL This is returned for requests that are invalid for one reason or another including:

- Attempting to write or write file-mark after reading (except in audio mode, for devices that support it) without an intervening close, or ioctl to re-position the tape.
- Attempting to read after writing (same exceptions as for writing).
- Using an invalid count on read, write, write file-mark, and so on.
- Attempting to do MTAFILE on a drive that doesn't support it (the MTCAN_APPEND bit is not set in the *master.d/scsi* file).
- Attempting to do an ioctl on a drive that doesn't support it (such as MTBSR on Cipher 540S) or attempting to do an unsupported MTOP operation or other unsupported ioctl's.
- Attempting to write to a QIC24 cartridge from a QIC150 drive (an MTIOCGET should be done, and the *QIC* bits should be checked for in mt_dposn to confirm this error).
- Attempting to do something when not at BOT that can only be done at BOT, such as writing or reading a Kennedy tape drive at a different speed than was previously used or switching from the variable block size device to the fixed block size device. (In IRIX 4.0, an attempt to use a different density is allowed, but the drive continues to use the original density.)

- Attempting to perform reads, writes, or ioctls other than MTIOCGET on the {**stat**} device.

EIO A generic error occurred, such as a SCSI bus reset, unrecoverable media error, and so on. Also occurs on close or read/write if the media has been unloaded while the device is open.

ENOMEM
An attempt was made to read data with a count less than that at which the block was written. This can only happen with drives that support variable block sizes. It can also occur if the kernel memory allocator is not able to allocate memory for the driver structures.

ENOSPC Occurs on read or space commands that encounter end of tape or end of data, on writes that are attempted at end of tape, and also on some other commands that encounter EOT or EOD.

EROFS A write or write file-mark was attempted to a write-protected tape.

ENODEV An open was attempted on a device with an invalid SCSI controller or SCSI ID specified (that is, a mknod with the wrong arguments was used to create the device special file) or an attempt to open a tape in variable blocksize mode when the device doesn't support it. Also occurs when the SCSI inquiry command fails or returns indicating that the device is not a tape device (media not removable, or not a sequential access device).

NOTES

High density tape cartridges such as the DC6150 (originally called 600 XTD) written on a system equipped with QIC 150 tape drives can NOT be read on older systems. Even if a low density tape (such as DC 600A) is used, it is still written at a higher density (QIC 120) than older tape drives can read. Tapes written on the older systems can still be read on the new tape drives, however. Systems with QIC 150 cartridge tape drives such as the Personal IRIS are able to read QIC24 tapes (310 oersted) such as the DC 300XL, but are not able to write them.

All tape devices other than the QIC (quarter-inch) tape drives have */dev/tape* linked to the {**ns**} device for performance, since there is no compatibility issue, and byte swapping is done in software. Most newer drives support variable block size devices, and the */dev/tape* link uses those by default; when multiple densities are supported, as with 9-track, the link is to the highest capacity device. For compatibility with earlier IRIX releases, the 8mm device is linked to the fixed block device. See the (unfortunately somewhat confusing) script */dev/MAKEDEV* for details; in particular, look at the **tapelinks** target.

Each time the tape drive is closed and the drive has reported recovered error, the driver reports to the console and (as normally configured) to */var/adm/SYSLOG* the number of recovered errors, if any. A small number is not indicative of problems, but a large number (somewhere above about 2-5% errors as a percentage of I/Os, depending on media age and quality) probably indicates that the media is approaching the end of its lifetime, that the drive read/write heads are dirty, or that the heads need to be realigned. A typical recovered error message might look like:

```
NOTICE: SCSI tape #0,7 had 8 successfully retried commands (1% of r/w)
```

The first number is the controller number, the second is the SCSI ID on that controller.

It is important to realize that these are *recovered* errors (at the drive level) and therefore do not result in errors being reported to the program doing the tape I/O.

FILES

/dev/rmt/tps*
/dev/rmt/jag*
/dev/mt exists as a symlink to */dev/rmt*; the use of the */dev/mt* pathname is deprecated and is supported only for compatibility
/dev/tape, /dev/nrtape, /dev/tapens, /dev/nrtapens
convenience links to the "preferred" device in */dev/rmt* (highest SCSI ID on lowest numbered SCSI bus, for tps devices only)
/var/sysgen/master.d/scsi
contains a configuration table indicating what devices support what features and what string should be matched against the string returned by the SCSI inquiry command and the *hinv*(1M) command; this was formerly in *master.d/tpsc*
/dev/MAKEDEV
a makefile (normally invoked by the superuser in the */dev* directory only) that creates devices that match the installed tape drives if invoked as **cd /dev; ./MAKEDEV tape**

SEE ALSO

MAKEDEV(1M), bru(1), cpio(1), hinv(1M), mt(1), tar(1), ioctl(2), rmtops(3), datframe(4), mtio(7).

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2159-005.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389