Selected IRIX™ Site Administration Reference Pages

Document Number 007-2159-004

Selected IRIX Site Administration Reference Pages
Document Number 007-2159-004

Silicon Graphics, Inc.
Mountain View, California

Introduction

This volume contains selected IRIX Site Administration Man Pages. The table below lists the Man Pages and summarizes their functions.

Man Page	Function
Add_disk(1)	add an optional disk to the system
autoconfig(1m)	configure kernel
Backup(1)	backup the specified file or directory
bootp(1m)	server for Internet Bootstrap Protocol
brc(1m)	system initialization procedures
chkconfig(1m)	configuration state checker
chroot(1m)	change root directory for a command
core(4)	format of core image file
cshrc(4)	system-wide csh initialization command file
devnm(1m)	device name
distcp(1m)	copy or compare software distributions
dvhtool(1m)	modify and obtain disk volume header information
ecc(1)	dump memory ecc log
ed(1)	text editor
find(1)	find files

Selected IRIX Site Administration Reference Pages

Man Page	Function
fs(4)	layout of the Extent file system
fsck, dfsck(1m)	check and repair file systems
fsdb(1m)	file system debugger
fsstat(1m)	report file system status
fstab(4)	static information about filesystems
ftp(1c)	Internet file transfer program
fx(1m)	disk utility
gettydefs(4)	speed and terminal settings used by getty
growfs(1m)	expand a filesystem
hinv(1m)	hardware inventory command
hosts(4)	host name-address database
init(1m)	process control initialization
inittab(4)	script for the init process
inode(4)	format of an Extent File System inode
inst(1m)	software installation tool
killall(1m)	kill named processes
lboot(1m)	configure bootable kernel
login(1)	sign on
lvck(1m)	check and restore consistency of logical volumes
lvinfo(1m)	print information about active logical volumes
lvinit(1m)	initialize logical volume devices
lvtab(4)	information about logical volumes
MAKEDEV(1m)	Create device special files
master(4)	master configuration database
mkboottape(1m)	make a boot tape

Selected IRIX Site Administration Reference Pages (continued)

Man Page	Function
mkfs(1m)	construct a file system
mklv(1m)	construct or extend a logical volume
mload(4)	dynamically loadable kernel modules
mount, umount(1m)	mount and dismount file systems
mtune(4)	default system tunable parameters
nvram(1m)	get or set non-volatile RAM variable(s)
passwd(4)	password file
profile(4)	setting up an environment at login time
prom(1m)	PROM monitor
prtvtoc(1m)	print disk volume header information.
ps(1)	report process status
pwck(1m)	password file checker
pwconv(1m)	install and update /etc/shadow with information from /etc/passwd
rcp(1c)	remote file copy
Restore(1)	restore the specified file or directory from tape
restore(1m)	incremental file system restore
rlogin(1c)	remote login
savecore(1m)	save a crash vmcore dump of the operating system
setmnt(1m)	establish mount table
sh(1)	shell, the standard command programming language
shadow(4)	shadow password file
statd(1m)	network status monitor daemon
stune(4)	local settings for system tunable parameters
su(1m)	become super-user or another user

Selected IRIX Site Administration Reference Pages (continued)

Man Page	Function
symmon(1m)	kernel symbolic debugger
syslogd(1m)	log systems messages
system(4)	system configuration information directory
systune(1m)	display and set tunable parameters
sys_id(4)	system identification (hostname) file
telnet(1c)	User interface to the TELNET protocol
ttytype(4)	data base of terminal types by port
versions(1m)	software versions tool

Selected IRIX Site Administration Reference Pages (continued)

NAME

Add_disk – add an optional disk to the system

SYNOPSIS

Add_disk [*controller number*] [*disk number*]

DESCRIPTION

The *Add_disk* command allows the user to add an extra SCSI disk to the system on integral SCSI controllers only (i.e., it many not be used for disks attached to the VME SCSI controllers).

The *disk number* option must be specified if not adding the default ID of 2; similarly the controller 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), and if they are present, the user is asked if they want to proceed, before a filesystem is made. If the answer is no, Add_disk exits.

NOTE

Older versions of this script 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

mkfs(1m), fx(1m), fstab(4)

NAME

autoconfig – configure kernel

SYNOPSIS

/etc/autoconfig [–**vf**] [–**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 start-up script in */etc/init.d*.

The options are as follows:

–**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.

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.

SEE ALSO

lboot(1m), setsym(1m), rc0(1m), rc2(1m)

NAME

Backup – backup the specified file or directory

SYNOPSIS

Backup [*–h hostname*] [*–t tapedevice*] [*–i*] [*directory name* | *file name*]

DESCRIPTION

The *Backup* command archives the named file or directory ("." if none 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**.

If a tape drive attached to a remote host is used for backup, the name of the remote host needs to be specified with the –**h hostname** option on the command line. For remote backup to successfully work, the user should have a TCP/IP network connection to the remote host and also have "guest" login privileges on that host.

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**.

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

The Backup command uses **bru** 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

Restore(1), List_tape(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:

   The requester's network hardware address
   The requester's Internet address (optional)
   The desired server's name (optional)
   The boot file name (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 will respond.

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 will look 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 will check 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 will not respond to the request.

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

   The requester's Internet address
   The server's Internet address
   The Internet address of a gateway to the server

The server's name
Vendor specific information (not defined by the protocol)

If the boot file is missing, the server will return a response packet with a null filename, but only if the request was specifically directed to that server. The pathname translation is: if the boot filename is rooted, use it as is; else concatenate the root of the boot subtree, as specified by the BOOTP configuration file, followed by the filename supplied by the requester, followed by a period and the requester's hostname. If that file is not present, remove the trailing period and host name and try again. If no boot filename is requested, use the default boot file for that host from the configuration table. If there is no default specified for that host, use the general default boot filename, first with *.hostname* as a suffix and then without. Note that *tftpd*(1M) must be configured to allow access to the boot file (see the *tftpd* manual entry 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 on ''Booting Through Gateways'' for an explanation.

Bootp Configuration File

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

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

# Default bootfile:
unix

%%

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

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

The fields of each line may be separated by variable amounts of white space (blanks and tabs). The first section, up to the line beginning '%%', defines the place where *bootp* looks for boot files when the client requests a boot file using a non-rooted pathname. The second section of the file is used for mapping client 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, which indicates that the hardware address is a 48-bit Ethernet address. The *haddr* field is the Ethernet address of the system in question expressed as 6 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 will select the first of the following that it finds:

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

The length of the boot file name 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* will only respond to a request from a client if it can deduce the client's Internet address. There are three ways that this can happen: 1) the client already knows its Internet address and includes it in the BOOTP request packet, 2) there is an entry in */usr/etc/bootptab* that matches the client's network hardware address or, 3) 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 will only be 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* will resend any BOOTP request it receives that asks for a specific host by name, if that host is on a different network from the client that sent the request. The BOOTP server forwards the packet using the full routing capabilities of the underlying IP layer in the kernel, so the forwarded packet will automatically be routed to the requested BOOTP server 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 mean that a system call has returned an error unexpectedly. Such errors usually cause *bootp* to terminate. The *message* will be 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 may be given as a host name or an Internet address.

request from xxx for ’fff’

> Information only. *Bootp* logs each request for a boot file. The 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 isn’t accessible.

reply boot file name fff too long

> The file name 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 (i.e., 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’s 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*(3) 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 file system. If the root file system is found to be bad, *bcheckrc* repairs it.

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

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

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

SEE ALSO

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

NAME

chkconfig – configuration state checker

SYNOPSIS

/sbin/chkconfig [–**s**]
/sbin/chkconfig *flag*
/sbin/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*) will not be run upon login, and therefore programs started from that file will not be run, or run with different options.

directoryserver	Start the Cadmin directory server daemon.
gated	Start Cornell routing daemon instead of BSD routed.
hypernet	Initialize HyperNET controller and routes.
ipfilterd	Enable SGI 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 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 network without removing cables.
nfs	Start the NFS daemons *nfsd* and *biod*. Mount all NFS filesystems.
noiconlogin	Don't 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 which have quotas enabled. See *quotas*(4) for more details.
quotas	Enable quotas for local configured filesystems.
rarpd	Start the Reverse ARP daemon.
routed	Start 4.3BSD RIP routing daemon. See *routed*(1M) for more details.
rtnetd	Initialize preemptable networking for real-time use.
rwhod	Start 4.3BSD rwho daemon.
sar	Start the system activity reporter.
snmpd	Start Simple Network Management Protocol daemon.
soundscheme	Start the Indigo Magic audio cue daemon.
timed	Start 4.3BSD time synchronization daemon.
timeslave	Start SGI time synchronization daemon.

verbose	Print name 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 may 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** , then 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 windowsystem are the commands *startgfx(1G)* and *stopgfx(1G).*
xdm	Start the X display manager.
yp	Enable NIS, start 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

/sbin/chroot newroot command

DESCRIPTION

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

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

```
chroot newroot command >x
```

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

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

This command can be run only by the superuser.

CAVEAT

In order to execute programs which 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. At the very least, it should contain a version of the libc.so shared object file. If you intend to run IRIX4 binaries, you should also copy /lib/libc_s into this directory.
./dev	The run-time loader needs the zero device in order to work correctly. Copy /dev/zero into this directory and make it read-only (mode 444).

SEE ALSO

cd(1), chroot(2)

NOTES

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

When using *chroot,* do not exec a command that uses shared libraries. This will result in killing your process.

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 will 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' stack, and data sections are normally written in the core image. The process' text is not normally written in the core image.

NOTE

Core image format designed by Silicon Graphics, Inc.

SEE ALSO

edge(1), dbx(1), ps(1), setuid(2), signal(2).

NAME

cshrc – system-wide csh initialization command file

DESCRIPTION

The file **/etc/cshrc** contains a list of commands to be invoked whenever a user logs into the system with *csh*(1) as their login shell. These commands are executed before those in the **.cshrc** and **.login** files in the home directory of the user.

FILES

/etc/cshrc

SEE ALSO

csh(1).

NAME

devnm – device name

SYNOPSIS

/etc/devnm [names]

DESCRIPTION

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

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

EXAMPLE

The command:

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

produces

```
/dev/dsk/ips0d0s2 usr
```

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

FILES

/dev/dsk/*
/etc/mtab

SEE ALSO

brc(1M).

NAME

distcp – copy or compare software distributions

SYNOPSIS

distcp [–**cnrsvw**] *from to* [*file* ...]

DESCRIPTION

distcp copies or compares *software distributions*. Software distributions are software releases for one or more software products that are prepared by Silicon Graphics and installed by *inst*(1M). *distcp* is typically used to create more copies of a software release tape, or to copy software from tape to a server workstation which becomes the software distribution source for many workstations on a network.

from is the location of the software distribution to be copied and *to* is the location where the copy will be created. *from* can be a tape device, a directory containing a software distribution, or the name of a product in a distribution directory. *to* can be a tape device or a directory. *from* and *to* can include the name of a remote machine and possibly a userid.

When accessing a remote machine, you must be superuser when you give the *distcp* command. In addition, the userid you use on the remote machine (default is ''guest'') must have read permission (for *from*) and/or write permission (for *to*). The exact syntax for *from* and *to* is identical to the syntax for the *source* argument of the *inst* –**f** option. See *inst*(1M) for details.

The optional *file* arguments are an additional way to specify what to copy or compare. A software distribution is a collection of *file*s. Some of these *file*s are archives that contain the files in the product while other *file*s contain information about what's in the software distribution, installation configuration information, and tools for performing the installation.

The possible *file*s are:

sa	**sa** contains the standalone tools and environment for miniroot installations (see *inst*(1M)).
mr	**mr** is an additional file for miniroot installations.
product	This file is known as a ''product descriptor'' for *product* and contains information about the contents of the distribution. *product* is a short name for one software product.
*product***.idb**	This file is called the ''idb file'' and contains one line for every file, directory, link and fifo in a software product.
product.image	These files are called ''images'' and contain the files that will be installed by *inst* on a workstation. Typical *image*s are 'sw' and 'man'.

A software distribution can contain multiple *product*s. The **sa** and **mr** files are required for doing miniroot installations, but need not be included in every software distribution you create with *distcp*. The default is to include them in copies or comparisons. The –**n** option can be used to exclude them.

By default, *distcp* copies software distributions. Options allow you to compare distributions or otherwise alter *distcp*'s default behavior:

–**c** Compare (rather than copy) *from* and *to*.

–**n** Do not include the standalone files **sa** and **mr**. The tape will not be bootable. The file **mr** is normally a 0 length file; it is required on bootable tapes for backwards compatibility. distcp no longers requires an empty **mr** file in a source directory when writing a bootable tape; it will create the empty tape file without it, when this option is not specifed.

–**r** Retension tape before reading or writing.

–**s** Compare silently; return exit status only.

–**v** Verbose; report *file* names as they are copied.

–**w** Warnings for short files; report during comparison if file sizes do not match.

EXAMPLES

To create a distribution directory that will enable users on a network to install from disk rather than tape, first create a directory (such as */d/newrel*) on a system that has enough disk space to contain all of the software on all of the tapes. Then, insert each tape in the drive and give this command once per tape:

```
distcp /dev/nrtape /d/newrel
```

To make a tape from a remote machine of just one product in a distribution directory, say the Network File System (nfs), give the command:

```
distcp –n machine:/d/newrel /dev/nrtape "nfs*"
```

NOTES

If you are using a tape to copy the distribution, it should be the no-rewind tape device. The tape device MUST also be the fixed blocksize device (and must be 512 byte blocks; use the *mt setblksiz 512* command to set the blocksize, if necessary) to work reliably, and to work at all for booting the miniroot it must be the byte swapped device. See *intro*(7) for more information on tape device names.

Only QIC (quarter inch) tapes are bootable on all machines. Some machines that are newer can boot from other tape types, but this is not guaranteed.

It is possible to copy to a remote directory but you cannot create a tape on a remote machine.

distcp can not be used to copy distributions from CD-ROM to a directory, since the CD-ROM is itself mounted as an EFS filesystem (one may of course use the mounted CD-ROM directory as a source when using *distcp* to copy to tape). To make a local copy of a CD-ROM, use **cp** with the **-r** option, or one of the many other methods of doing directory copies. Similarly, to copy an installable CD-ROM over the network, use **rcp** with the **-r**

distcp takes **tape** as an argument by itself to be a synonym for */dev/nrtape*. Thus, if you want to refer to a directory named **tape**, you must refer to it by the full pathname, or **./tape**, or similar workarounds.

SEE ALSO

inst(1M), versions(1M), cp(1), rcp(1), tps(7)
IRIS-4D Series Installation Guide.

NAME

dvhtool – modify and obtain disk volume header information

SYNOPSIS

/sbin/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.

Note that it is necessary to be superuser in order to use *dvhtool.*

Invoked with no arguments (or just a volume header name), *dvhtool* allows the user to interactively examine and modify the disk volume header on the root drive. The **read** command prompts for the name of the device file for the volume header to be worked on. This may 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: 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's* command line arguments.

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

The –**v** flag 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**.

The **creat** option allows creation of a volume directory entry with the name *dvh_file* and the contents of *unix_file.* If an entry already exists with the name *dvh_file,* it is overwritten with the new contents.

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

The **delete** option removes the entry named *dvh_file,* if it exists, from the volume directory.

The **get** option copies the requested file from the volume header to the file system.

The **list** option lists the current volume directory contents.

SEE ALSO

vh(7m)
fx(1m)

NOTE

Several Mbytes 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, since all files must then be copied to /tmp, and then back to the volume header.

NAME

ecc – dump memory ecc log

SYNOPSIS

ecc [-c]

DESCRIPTION

ecc dumps the memory ecc log. This log is produced by the memory subsystem each time a memory error occurs. 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 will 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 as to future more serious problems. The -*c* option will cause the log to be cleared.

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

Note that on system crash and subsequent reset the log contents are not cleared, and may be read via the prom monitor. 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.

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. 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** on **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 may 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 manual 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 will only allow 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 will automatically be used when scanning *file*. For example, if the first line of a file contained:

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

tab stops would be set at columns 5, 10, and 15, and a maximum line length of 72 would be imposed. NOTE: 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 may 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:

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 new-line.

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 new-line and the remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (–) may 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 may 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 may 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 **\(** and **\)** is an regular expression that matches whatever the unadorned regular expression matches.

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 may 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 may 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 –, then 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 may 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**) may 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 may be entered from a terminal is 256 per line (including the new-line 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 file name, 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 file name 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 file name. See also DIAGNOSTICS below. If *file* is replaced by **%**, and if additional *file* arguments were specified on the command line, then the next file name 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 file name to *file*; otherwise, it prints the currently remembered file name.

(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 may 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) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a new-line 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 may 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 will also explain 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 may be entered from a terminal is 256 per line (including the new-line character).

(.,.+1)j
The **j**oin command joins contiguous lines by removing the appropriate new-line 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 may 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 may 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 may 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 will prompt 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 file name, if any, is used (see the **e** and **f** commands). The currently remembered file name is not changed unless *file* is the very first file name 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 file name.

`(.,.)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 new-line may 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 may 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 may be split by substituting a new-line character into it. The new-line 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 may 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 file name is not changed unless *file* is the very first file name mentioned since **ed** was invoked. If no file name is given, the currently remembered file name, 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 file name.

(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 will 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 file name; if a **!** appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, **!!** will repeat the last shell command. If any expansion is performed, the expanded line is echoed; the current line (**.**) is unchanged.

(.+1)<new-line>
An address alone on a line causes the addressed line to be printed. A new-line 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 new-line 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 new-line, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

lf4w(.75i) lf4. 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, then **/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** on **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

? for command errors. Type the ‘h’ command for a short error message.
?*file* for 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 will take 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)].

The encryption options and commands are provided with the Encryption Utilities package, which is available only in the United States.

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 new-line 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 path name in the *path-name-list* (that is, one or more path names) seeking files that match a boolean *expression* written in the primaries given below. 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 file name. Normal shell argument syntax may be used if escaped (watch out for **[**, **?** and ∗).
–**perm** [–]*onum*	True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, only the bits that are set in *onum* 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 has been accessed in *n* days (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 has been modified in *n* days (see *stat*(2) for a description of which file operations change the modification time of a file).
–**ctime** *[***+-***]n*	True if the file has been changed in *n* days (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 path name.
–**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 path name to be printed.
–**cpio** *device*	Always true; write the current file on *device* in *cpio* (1) format (5120-byte records).
–**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 path name is a directory, *find* will not descend into that directory, provided –**depth** is not also used.
–**mount**	Always true; restricts the search to the file system 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 file systems.
–**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 may be combined using the following operators (in order of decreasing precedence):

1) The negation of a primary (**!** is the unary *not* operator).

2) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).

3) 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 file system 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), fs(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`** will fail at that point and issue some number of ‘‘stat failed:’’ messages.
`-depth` and **`-prune`** do not work together well.

NAME

efs – layout of the Extent file system

SYNOPSIS

#include <sys/param.h>
#include <sys/fs/efs.h>

DESCRIPTION

An Extent filesystem may 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 long sectors, also called *basic blocks*.

Note that the current maximum size limit of an Extent filesystem is 16777214 blocks, equivalent to 8 gigabytes.

The Extent filesystem 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 *super-block*. The format of an Extent file system super-block is:

```
/*
 * Structure of the super-block for the Extent file system
 */
struct  efs {
      /*
       * This portion is read off the volume
       */
       long    fs_size;               /* size of file system, 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 file system */
       short   fs_dirty;              /* fs needs to be fsck'd */
       time_t  fs_time;               /* last super-block update */
       long    fs_magic;      /* magic number */
       char    fs_fname[6];   /* file system name */
       char    fs_fpack[6];   /* file system 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.
         */
};
```

Note that 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 file system. This variable is filled in when the file system 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 file system. 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 file system. Note that *fs_heads* is in fact currently unused, and should not be relied upon.

fs_dirty is a flag which indicates if the file system needs to be checked by the *fsck*(1M) program. The *fs_time* field contains the time stamp of when the file system was last modified. *fs_name* holds the *name* of the file system (where it is mounted, more or less) while *fs_fpack* contains which volume this file system is. The *fs_fpack* field is singularly useless, but is provided for utility compatibility. *fs_magic* is used to tag the superblock of the file system 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 *Imkfs -r* or extended with *growfs* will get the new magic number; otherwise the older magic number will be 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 file system has been constructed on a logical volume which 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 file system, 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, then the file system must be cleaned with *fsck* before it can be mounted. If the file system is the *root* partition, then 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), mkfs(1M), growfs(1M), inode(4).

NAME

fsck, dfsck – check and repair file systems

SYNOPSIS

/etc/fsck [–**c**] [–**f**] [–**g**] [–**m**] [–**n**] [–**q**] [–**y**] [–**l** **dir**] [file-systems]
/etc/dfsck [options1] fsys1 **...** – [options2] fsys2 **...**

DESCRIPTION

Fsck

fsck audits and repairs inconsistent conditions for file systems. The user must have both read and write permission for the device containing the file system, unless the **-n** flag is given, in which case only read permission is required.

If the file system is inconsistent, the user is normally prompted for concurrence before each correction is attempted. It should be noted that most corrective actions will result in some loss of data. The amount and severity of data loss may be determined from the diagnostic output. The default action for each correction is to wait for the user to respond **yes** or **no**. 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 file system only if the superblock indicates that it is dirty, otherwise a message is printed saying that the file system is clean and no check is performed. The default in the absence of this option is to always perform the check.

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

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

–**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.

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

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

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

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

If no *file-systems* are specified, *fsck* will read a list of default file systems from the file **/sbin/fstab**. Note that this will not include the root file system, *fsck* will run on root only if this is explicitly specified.

Normally, a file system must be unmounted in order to run *fsck* on it, an error message is printed and no action taken if invoked on a mounted file system. The one exception to this is the root file system, 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 file system parameters, *fsck* will scan **/etc/fstab** and attempt to fork a check process for each **efs** file system found. These checks proceed in parallel, for improved speed.

The name of the device holding the file system 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 file system resides. For example, if a file system was on */dev/dsk/ips0d1s7* the logfile would be named */etc/fscklogs/ips0d1s7*.

Note that since 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 file system 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 file systems, they will be checked in the order of their pass numbers. All file systems with a given pass number will be checked (in parallel, if more than one file system 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 file systems will be checked simultaneously if possible.

(Note: in fact, *fsck* takes note of the amount of memory available in the system, and will limit the number of simultaneous check processes to avoid swapping).

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 file system, 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 file system 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 user's concurrence, reconnected by placing them in the **lost+found** directory, if the files are nonempty. The user will be notified if the file or directory is empty or not. Empty files or directories are removed, as long as the −**n** option is not specified. *fsck* will force the reconnection of nonempty directories. The name assigned is the i-node number. The directory **lost+found** must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This directory is always created by *mkfs(1m)* when a file system is first created.

SUPERBLOCKS AND FILESYSTEM ROBUSTNESS

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

Finally, if no superblock can be found on a damaged file system, it may be possible to regenerate one by using the new *-r* option of *mkfs(1M)*, and then use *fsck* to salvage the file system. Warning: this will **not** be effective if the file system 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 file system and errors were detected. The behavior now is always to remount the root file system 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 will always be rebuilt if necessary.

All of these options are now legal no-ops.

DFSCK

Note: now that *fsck* has the capability of checking file systems in parallel, use of *dfsck* is strongly deprecated. This program will not be supported beyond the current release.

Dfsck should not be used to check the *root* file system.

Dfsck allows two file system checks on two different drives simultaneously.

options1 and *options2* are used to pass options to *fsck* for the two sets of file systems. A – is the separator between the file system groups.

The *dfsck* program permits a user to interact with two *fsck* programs at once. To aid in this, *dfsck* will print the file system name for each message to the user. When answering a question from *dfsck*, the user must prefix the response with a **1** or a **2** (indicating that the answer refers to the first or second file system group).

FILES

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

SEE ALSO

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

NAME

fsdb – file system debugger

SYNOPSIS

/etc/fsdb [**options**] special

DESCRIPTION

The options available to **fsdb** are:

-?	display usage
-o	override some error conditions
-p'string'	set prompt to string
-w	open for write

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

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

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 should be the name of a character device file. **fsdb** searches /etc/fstab for the raw character device file name, if given the name of a file system. 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-through's will bypass the file system 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 file system. 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 is to be displayed or accepted. The *base* command will display or set the input/output base. Once set, all input will default to this base and all output will be shown in this base. The base can be overridden temporarily for input by preceding hexadecimal numbers with '0x', preceding 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** will record 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
```

will first set the value of *dot* to 2, ':' will alert the start of a command, and the *inode* command will set *inode* to 2. A count is specified after a ','. Once set, *count* will remain at this value until a new command is encountered, which will then reset the value back to 1 (the default). So, if

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

is typed, 400 hex longs are listed from 2000, and when completed, the value of *dot* will be 2000 + 400 * sizeof (long). If a carriage-return is then typed, the output routine will use the current values of *dot*, *count*, and *type* and display 400 more hex longs. A '*' will cause the entire block to be displayed.

End of block and file are maintained by **fsdb.** When displaying data as blocks, an error message will be 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 would be:

```
> 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 file system. Once there, subsequent carriage-returns (or +, -) will 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 may be composed of +, -, *, and % operators (evaluated left to right) and may use parentheses. Once evaluated, the value of *dot* is updated.

,*count* Count indicator. The global value of *count* will be updated to *count*. The value of *count* will remain until a new command is run. A count specifier of '*' will attempt to show a *blocks'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 don't 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 may 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 *UNIX* counterparts (see individual command for details). The '*', '?', and '[-]' wild card 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 file names whose i-number matches *i* or whose name matches pattern *n*. Note that only one of the two options (-name or -inum) may 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* will be 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 may be used (but, if used, *must* be specified before the file name specifiers). Also, as stated above, wild card characters are available and multiple arguments may 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 one of the **-R** or the **-l** options is used, then the files may have a character following the file name, indicating the type of the file. Directories have a "/", symbolic links have a "@", AF_UNIX address family sockets have a "=" and fifo's have "f". Regular files and block and character device files have a "*" if they are executable. If the file type is unknown, then a "?" is printed.

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

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

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 will set the value of *dot* to *the* superblock and display 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 may 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
```

would increment 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 will not be 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 - 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
```

would get the contents of disk block number field of extent number 1 from the inode, convert it to a block address. 20 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 - 11. This field is one byte long.

For example,

```
> 1:len/b
```

would display 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 will allow you to change or display the directory name. For example,

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

will get the 7th directory entry of the current *inode* and change its name to foo. Note that 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 - 11. This field is 3 bytes long. For example,

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

would display 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
Increments 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, due to 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 5th 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 file system.
> 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 will stop prematurely if the eof is reached.

SEE ALSO

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

NAME

fsstat – report file system status

SYNOPSIS

/sbin/fsstat special_file

DESCRIPTION

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

SEE ALSO

fs(4)

DIAGNOSTICS

The command has the following exit codes:

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

NAME

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 character indicates a comment.

The entries from this file are accessed using the routines in *getmntent*(3), which returns 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 */
        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 path name of the directory to be mounted on. The *mnt_type* field determines how the *mnt_fsname* and *mnt_opts* fields will be interpreted. Here is a list of the filesystem types currently supported, and the way each of them interprets these fields:

efs	*mnt_fsname* must be a block special device (e.g., /dev/root).
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 (e.g., /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** file system type.

dos	*mnt_fsname* must be a floppy device. These are located in the directory /dev/rdsk (e.g., /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 (e.g., /dev/rdsk/fds0d2.3.5hi). See *smfd*(7M). SCSI devices are located in the directory /dev/scsi (e.g., /dev/scsi/sc0d4l0). See *ds*(7M).
swap	*mnt_fsname* should be the full path name to the file or block device to be used as a swap resource.
cachefs	*mnt_fsname* should be the file system name for the backing file system to be mounted as a cache file system. This will either be the special file name (e.g., /dev/scsi/sc0d7l0) or host:path.

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 may 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**):

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 super-users. This option has no effect for the super-user.
nodev	access to character and block special files is disallowed.

Options specific to **efs** filesystems (the default is **fsck, noquota**):

raw=*path*	the filesystem's raw device pathname (e.g. /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 **NBPC** (as defined in */usr/include/sys/param.h*), and no larger than 32768. 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):

setx
: set execute permission on every file on the mounted file system. 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 semi-colon.

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 3 letters **c**, **l**, or **m** may 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 may 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:

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.
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)

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. Note that EFS uses caching strategies similar to private NFS, and that 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:

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. Note: the **backfstype** argument must be specified. Additionally, any mount points which share the same cache directory must have the same set of the following options: **write-around**, **non-shared**, **noconst**, and **purge**. For example, if one CacheFS file system is mounted with the options "cachedir=/foo,noconst", all mounts for the cache directory /foo must have the noconst option as well.

backfstype=*file_system_type*
The file system type of the back file system (for example, **nfs**). Any file system type may be used as the back file system with the exception of **proc**, **fd**, and **swap**.

backpath=*path*
Specifies where the back file system is already mounted. If this argument is not supplied, CacheFS determines a mount point for the back file system.

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. The **write-around** mode (the default) handles writes the same as NFS does; that is, writes are made to the back file system, and the affected file is purged from the cache. You can use the **non-shared** mode when you are sure that no one else will be writing to the cached file system. In this mode, all writes are made to both the front and the back file system, 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 file system read-only.

local-access
: Causes the front file system to interpret the mode bits used for access checking instead or having the back file system verify access permissions.

purge Purge any cached information for the specified file system.

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 file system has changed. If it has, all information about the file is purged from the cache and new data is retrieved from the back file system. 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 file system has changed. If it has, all information about the directory is purged from the cache and new data is retrieved from the back file system. 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*.

NOTES

The default *fstab* supplied with IRIS-4D systems 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* will 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 would work in normal operation, it will lead to incorrect behavior when installing new software.

FILES

/etc/fstab

SEE ALSO

swap(1), fsck(1M), mount(1M), quotacheck(1M), quotaon(1M), cfsadmin(1M), fsck_cachfs(1M), getmntent(3), fd(4), mtab(4), proc(4).

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. The program allows a user to transfer files to and from a remote network site.

The client host with which *ftp* is to communicate may be specified on the command line. If this is done, *ftp* will immediately attempt to establish a connection to an FTP server on that host; otherwise, *ftp* will enter its command interpreter and await instructions from the user. When *ftp* is awaiting commands from the user the prompt ''ftp>'' is provided 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 will be 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 file name 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 will also terminate the session and exit.

case Toggle remote computer file name case mapping during **mget** commands. When **case** is on (default is off), remote computer file names 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 sytem 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 may contain single linefeeds; when an ascii type transfer is made, these linefeeds may 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* will prompt 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 file name 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 file name arguments are taken literally and not expanded. Globbing for **mput** is done as in **csh**(1). For **mdelete** and **mget**, each remote file name 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 will 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* will prompt 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 may be specified. If interactive prompting is on, *ftp* will prompt 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 file name thus produced. See **glob** for details on the filename expansion. Resulting file names will then be 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 may be specified, and the *local-file* must be specified. If interactive prompting is on, *ftp* will prompt 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 file names will 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* will prompt 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 file name "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 may be included in *outpattern*, as in the 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 file name.

open *host* [*port*]

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

prompt

Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default is on), any **mget** or **mput** will transfer all files, and any **mdelete** will delete 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** will not define new macros during the auto-login process, **close** will 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, and **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 file name 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 will be reported. Note that **runique** will 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* will attempt 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* will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

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 file names. Remote ftp server must support ftp protocol STOU command for successful completion. The remote server will report 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* will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user will be prompted for it. If an account field is specified, an account command will be 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 which have embedded spaces may be quoted with quote (") marks.

ABORTING A FILE TRANSFER

To abort a file transfer, use the terminal interrupt key (usually Ctrl-C). Sending transfers will be immediately halted. Receiving transfers will be 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 will not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence will be ignored when *ftp* has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode may 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 file name ''−'' is specified, the **stdin** (for reading) or **stdout** (for writing) is used.

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

3) Failing the above checks, if ''globbing'' is enabled, local file names are expanded according to the rules used in the *csh*(1); c.f. the *glob* command. If the *ftp* command expects a single local file (e.g., **put**), only the first filename generated by the "globbing" operation is used.

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

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

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters which may affect a file transfer. The *type* may 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 may be specified at the shell command line. Several options can be enabled or disabled with *ftp* commands.

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

The –**n** option restrains *ftp* from attempting ''auto-login'' upon initial connection. If auto-login is enabled, *ftp* will check 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* will prompt 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.

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

The –**d** option enables debugging.

The –**g** option disables file name 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 may be separated by spaces, tabs, or new-lines:

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 will initiate a login using the specified name.

password *string*
: Supply a password. If this token is present, the auto-login process will supply 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* will abort 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 will supply the specified string if the remote server requires an additional account password, or the auto-login process will initiate an ACCT command if it does not. Note that if this token is present in the .netrc file, *ftp* will abort 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 new-line 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 [-x] [-r maxretries] [-l logfile] [-c] [drivetype[(ctrlr,unit)]] [VERIFY] [INITIALIZE] [FORMAT] (with -c)

DESCRIPTION

fx is an interactive, menu-driven disk utility. It allows bad blocks on a disk to be detected and mapped out. It also allows display of 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* will also warn of discrepancies between the disk label and the parameters that would normally be used for the drive, and then ask 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, since the changes will frequently result in a different partition layout.

The **-r** *retries* option allows you to specify how many retries *fx* will attempt when exercising the disk. If you have persistent soft errors, **-r** *0* will usually allow *fx* to find the bad sectors and spare them. For SCSI disks, also see the section on *parameters* under the **LABEL MENU**.

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

The **-c** option (in the UNIX 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 only creates 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 will create a new partition table (suitable for an option disk) and volume header, if necessary.

USING FX

There are 2 versions of *fx*. One runs in the standalone environment, and must be used when the system disk will be modified; it may 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 a normal command, and must normally be 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 mount disk, or a disk whose partitions are part of mounted logical volume, this version will warn 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 may be invoked when the system is not running by giving the following command at the PROM monitor:

```
boot stand/fx
```

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

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

Or, for later systems with the ARCS prom (R4K Indigo, Indigo2, Indy, Onyx, Challenge):

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

Or, for R8000 systems with 64-bit ARCS prom (POWER Challenge, POWER Onyx, R8000 Indigo2):

```
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, and CPU is the CPU type (for example, IP6). If CPU consists of 4 or more characters, omit the **.** (period) after *sash*. (Note that IP7, IP9, and IP15 are not really distinct CPU types from IP5; for all 4 of these models, use IP5 for CPU.)

fx may also be booted from tape by giving this command at the PROM monitor:

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

   or

boot -f tpqic()fx.CPU
```

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

The command version of *fx* is invoked by name like any regular 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, *dkip* for ESDI drives with Interphase controllers, *xyl* for SMD drives with Xylogics controllers, *ipi* for IPI drives with Xylogics controllers, *jag* for SCSI drives connected to the Jaguar VME SCSI controller *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 will normally be 0 unless your system has more than one controller and you wish to work on disks attached to an additional controller (1, 2, etc.). 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 may 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)" "drive_type"
```

For example:

```
fx "dkip(0,1)" "Hitachi 512-17"
```

or

```
fx "dksc(0,1)"
```

The quotes are necessary in the first argument in the command version, since 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 then performed, followed by sanity checks on the partition layout, etc., are performed. If any 'major' differences are found, you will be 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 will ask if you want to use the defaults; you may answer no if you plan to set up custom parameters or partitions.

fx then enters its main menu. Menu items may be selected by typing the least unambiguous prefix (the portion included between **[** and **]**, or the full name may be entered. A menu item may 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 will cause that submenu to be displayed, and items from it may then be selected. To return to a parent menu from a submenu, enter two dots (".."). The menus are organized as a hierarchy, so one may go up 2 levels by typing ../.., or use a command several levels down by separating each level by a `/`. By typing a "full pathname", such as

```
/label/show/partition
```

a command may be executed from any point in the menu hierarchy. Similarly, typing the full pathname of any menu will move 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 the item may 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 will occur if modified parameters are not yet committed to disk.

Once the main menu is reached, *fx* catches interrupts: an interrupt will stop any operation in progress but will not terminate *fx* itself. The current operation executing in the disk driver (if any) will complete first; this is most notable when formatting a SCSI disk, as 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 RETURN will accept that value. For other prompts that ask a question, you must answer either "yes" or "no". For prompts requesting numeric values, you may usually reply with a decimal number, or a hex number (a leading 0x). If a number is displayed at the end of the prompt, then pressing RETURN will accept 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 "?" will give 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 will give 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.

With non-SCSI disks, it is possible to format a range of cylinders; prompts are given for start cylinder and number of cylinders, the defaults being the entire disk. With SMD disks, the defect information placed on the disk by the manufacturer is destroyed by formatting. When formatting an SMD disk, *fx* automatically attempts to read and save this information first. If the disk has been previously formatted, *fx* will find no defect information on the disk. In this case, it will print a message saying that no defect information was found after trying a few tracks, offering the possibility of abandoning the search for it or continuing. If it is known that an SMD disk has been previously formatted, the reformat will be considerably speeded up by a "no" answer to this question.

BADBLOCK MENU

Badblocks are handled differently by SCSI and IPI/ESDI/SMD controllers. In the case of SCSI controllers, the list of badblocks is maintained by the SCSI controller/formatter hardware; 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.

For IPI/ESDI/SMD disks, the badblock list is a structure maintained explicitly by *fx* in the SGI-specific label area on the disk. IPI/ESDI/SMD badblocks are managed on a track basis: if a track has one or more bad blocks, the entire track is replaced with one from the "track replacement" area reserved on the disk.

The badblock menu contains the following choices:

addbb
: Allows new bad blocks to be added to the badblock list. Blocks may 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. In the IPI/ESDI/SMD case, it goes into the in-core copy of the badblock list maintained by *fx*, and does not become effective until the *forward* command has been given.

deletebb
: Deletes a block from the badblock list. This function does not appear for SCSI disks. There is no way to remove an added badblock from a SCSI disk without reformatting the whole disk.

  You will be 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: (1) to allow any mistake made during initial bad block entry to be corrected, and (2) to move data back off a replacement track which is also going bad.

  In the second case, be certain to use dd(1) to save the data onto another disk, delete the bad block forwarding ("deletebb" orig), add the replacement track to the bad block list ("addbb" replacement), update the forwarding ("forward"), mark the original block as bad ("addbb" orig), update the forwarding again ("forward"), and then use dd(1) to replace the saved data. This procedure is necessary to ensure data integrity, as data may be corrupted when moving it back onto the original defective track.

readdefects
: Appears only in the menu for non-SCSI disk types. It reads the defect information placed on the disk by the disk manufacturer, and adds bad blocks to the bad block list. (If the blocks identified by the manufacturer's defect information are already in the bad block list, nothing is altered). This function is useful for retrieving original defect information if the badblock list in the disk label has become lost or corrupted for some reason. Note that once an SMD disk has been formatted, the manufacturer's defect information is overwritten.

readinbb
: Reads the bad block list in the disk label into memory. (Note that *fx* does not keep an in-core bad block list for SCSI drives, so this function appears only for non-SCSI drive types). It should be used before adding any bad blocks, to ensure that the current badblock list is being worked on.

showbb
: Displays the current badblock list. For non-SCSI disks, the displayed list is the in-core copy kept by *fx*, originally read in with *readinbb* and possibly modified with *addbb*.
  For SCSI disks, it is obtained by interrogating the SCSI controller. 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).

forward

For non-SCSI disks, this saves the badblock list to disk, and causes the disk controller to map out any newly added bad blocks. (It does not appear for SCSI disks since added bad blocks on a SCSI disk are effective immediately).

The remaining option appears only in expert mode for non-SCSI disks.

createbb

Clears out any existing badblock list held by *fx* for the disk. It is used primarily when a disk is being completely reformatted.

Most disks have a number of defective spots where data cannot be stored. It is normally less than 1 defect per megabyte of disk capacity. Disks with higher numbers of defects may be failing, or may require formatting, particularly if new defects keep appearing. The disk driver or controller is able to get around this by dynamically replacing a bad block with a good block from a pool reserved for this purpose. A list is kept on the disk of defects and their replacements. 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 will print 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 will give 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 file name: see dkip(7), dksc(7), ipi(7), jag(7), rad(7) or xyl(7).

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 badblock operations. Badblock 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 will attempt to read each block in a failed i/o individually to find the bad block(s).

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* will be 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 will only add 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 block(s). However, if the disk is backed up and can be restored after the exercise is complete, a write-compare exercise pass will find and automatically map 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, as the badblock map is maintained by the drive itself, rather than by the driver. Note that *rad* devices do not support badblock forwarding. It is not necessary to read a badblock list before adding new bad blocks.

For non-SCSI disks, select the *readinbb* item to read in the existing badblock list from the disk. Next, select the *showbb* item to display the existing badblock list. Typically, there will be a small number of entries. If there are no entries, it is possible that the badblock list has been lost or corrupted. In this case, you should attempt to recover the manufacturer's original defect list by entering *readdefects*.

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 badblock can be entered. When you have finished entering, terminate the entries by entering two dots (".."). The updated badblock list must then be saved to disk and the new bad blocks mapped out. Select the *forward* option on the bad-block 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 badblocks are entered by logical block number relative to the start of the disk, using the *addbb* function. (Driver messages about bad blocks typically give 2 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 badblocks as you want, one per line, ending the list by typing ".." on a line by itself. The *showbb* function will display the complete list of bad blocks. The **-m** option may be used to show only the manufacturer's bad block list, in one of several formats, which varies from drive to drive; the default and the most common 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 bad block 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 part(s) of the label you wish to display. This will read 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 then 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, since 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 return for every prompt will leave the values unchanged. For SCSI drives, the drive parameters are divided into *geometry* and *parameters* menus. Changes to the *geometry* values will 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 would normally be 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 will depend on the type of controller. For SCSI disks, there is an additional menu called "geometry", and changes to values on the parameter menu will not require a reformat of the drive; changes to those on the geometry menu will.

These values will 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 will 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 will require reformatting the drive before it can be used. Not all drives support changing all geometry items. Some changes will also affect drive capacity. For some drives this capacity change will be reflected immediately in values read from the drive, while for others the new values will not be returned until after the drive is formatted.

partitions

The disk surface is divided for convenience into a number of different sections ("partitions") 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 file name 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 may be displayed and changed by using the *repartition* menu; See the section, **CHANGING DISK PARTITIONS**.

There may 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, as 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, file system, disk label, swap, and so forth). The **MAKEDEV** program will only create 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 would remove 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 will write the *sgilabel* file when it has changed and the user requests it. Also note that using *create/directory* will clear 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 than 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 file systems on partitions before you can install software or restore files onto them.

The *rootdrive* function will create 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 will create a drive with all of the usable area in a single partition (partition 7). Some space is still allocated to the volume label, and for non-SCSI drives, to the badblock replacement partition.

The *resize* function allows you to resize any of the standard partitions (root, swap, usr, and entire). After you select this function, a message is shown, reminding you that after you finish resizing a partition, the other partitions will be resized to match (if necessary). You will be 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 then 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, except for drive types that require the entire track to be mapped (such as ESDI drives).

For each unrecoverable error that is found, the failing block is added to the bad block 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, will do retries before reporting an error. For most SCSI drives, the number of retries performed may be set by using the */label/set/param* menu. By using the *stoponerror* menu selection, you can have *fx* stop and ask you if you want to map the badblock. Whether you answer yes or no, you are then 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 is intended to stress the head positioning system of the drive, and will sometimes find 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 may be executed in a number of different modes (read-only, read-write etc) that will be 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/o's (from 1 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 sometimes 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, then you are asked on each error whether you want to continue or not. If you continue, you are then asked if you want to add the failing block to the bad block list. This can be useful if you want to find all the failing bad blocks but not actually add them to the bad block 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 will be used in tests which write to the disk to be created. Up to 4Kbytes of pattern may be set, byte-by-byte. Each byte may be entered as a decimal or hex value (with a leading 0x). Enter a .. when you are done entering the pattern. The pattern will be 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 will have the same data. This helps find drives that have hardware or firmware problems causing them to write data to the wrong location on the drive, when used with the write-compare test.

showtestpat
: Displays the pattern of data that will be used in tests which write to the disk. This may 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 will occur 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 block(s) 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, since 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 will 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 will still cause 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 may be written as well as read. For SCSI disks, the drive parameters (modesense pages) may be displayed, and individual bytes can be altered, and then sent to the disk via modeselect commands.

A function that may 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 may 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 may be cleared by entering a null string).

number
: Accepts a decimal number, and prints it in octal and hex. For non-SCSI drive types only, it also prints the input, interpreted as a blocknumber in the form of cylinder/head/sector for the current drive. It will also accept input in the form of cylinder/head/sector (for example, 123/4/5) and print the corresponding blocknumber. (SCSI drives are always accessed by a single blocknumber, so this translation is not performed if the current drive type is SCSI).

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 may be read in one operation. The disk block address from which the read will occur 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. Does not cause any i/o, just sets the block number for the next i/o.

The remaining functions appear only in expert mode, since 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*.

showuib
: Appears only for non-SCSI drives. It prints the Unit Initialization parameters used by the disk controller for the current drive.

rawreadbuf
: Appears only for non-SCSI drives. It is something of a misnomer, because it actually invokes the disk controller command intended for reading non-SCSI manufacturer's flawmaps at the start of tracks. It prompts for destination buffer address, disk address and length. The sector part of disk address is irrelevant, because the controller always reads at start of track for this function. Also, the length parameter is ignored! The controller always transfers one sector for this operation.

passthru
: Appears only because the runtime *fx* utility shares common code with the standalone version. Direct passthrough of drive command codes is not allowed at runtime.

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 may 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 will be 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 may 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, as this could damage the drive or controller. On SCSI drives, it could also cause the termination power fuse to fail (on those machines 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 will be determined automatically by an inquiry operation on the drive. For non-SCSI drives, a menu of known drive types will appear. It is important to know the exact model number of the drive you are adding.

It is possible to work with (non-SCSI) drives other than those on the menu by entering a drive name of "other", and then entering parameters for the drive. This is intended only for engineering tests and evaluations. 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* will choose 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 will format the drive, scan it for bad blocks, and place a label on it. On completion, exit from *fx*. The drive is now ready for use.

It will usually be necessary to make file systems on the drive and to mount these file systems 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 will format 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 will work, but may not give you the same drive capacity or performance as those shipped by Silicon Graphics.

FILES

/dev/rdsk/jag*, /dev/rdsk/ipi*, /dev/rdsk/ips*, /dev/rdsk/dks*, /dev/rdsk/xyl*, /dev/rdsk/fds*, /dev/rdsk/rad*

SEE ALSO

mknod(1M), *mount*(1M), *dvhtool*(1M), *MAKEDEV*(1M), *Add_disk*(1), *xyl*(7), *dkip*(7), *dksc*(7), *rad*(7), *ipi*(7), *smfd*(7), *jag*(7), and *vh*(7).

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.A.) 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**, etc., 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*(1M) 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*(1M). The flags that *getty*(1M) 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*(1M) 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*(1M) executes *login*(1).
final-flags	These flags take the same values as the *initial-flags* and are set just before *getty*(1M) executes *login*(1). 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 new-line), they are included in the *login-prompt* field. As a special feature, this field may contain the string **$HOSTNAME** which will be replaced by the current host name 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*(1M) will search for the entry with *next-label* as its *label* field and set 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 in turn is linked to **300**, which finally is linked to **2400**.

If *getty*(1M) 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*(1M) can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into *getty*(1M) which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through *getty*(1M) with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

login(1), stty(1), getty(1M), ioctl(2), termio(7)

WARNINGS

To support terminals that pass 8 bits to the system (also, 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 will permit 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*(1M) 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

growfs – expand a filesystem

SYNOPSIS

/sbin/growfs [-s size] special

DESCRIPTION

Growfs expands an existing Extent Filesystem, see *fs(4).* The *special* argument should be 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 *size* argument is given, the filesystem will be grown to occupy *size* basic blocks of storage (if available).

If no *size* argument is given, the filesystem will be grown to occupy all the space available on the device.

It is anticipated that *growfs* will most often be used in conjunction with *logical volumes* see *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).*

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* is used for this.

In either case *growfs* is run on the logical volume device, and the expanded filesystem is then available for use on the logical volume device.

DIAGNOSTICS

Growfs will expand 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 may 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 will not recognize it.

SEE ALSO

mkfs(1M), mklv(1M), lv(7M).

NAME

hinv – hardware inventory command

SYNOPSIS

hinv [**-v**] [**-s**] [**-c** class] [**-t** type]

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 will display a one line description of each entry in the table. The **-v** option will give a more verbose description of some items in the table. The **-c** *class* option will display items from *class.* Classes are *processor, disk, memory, serial, parallel, tape, graphics,* and *network.* The **-t** *type* option will display items from *type.* Types are *cpu, fpu, dcache, icache, memory,* and *qic.* The **-s** option, when used with either the **-c** or **-t** options, suppresses output.

The hinv command, when used with the **-c** or **-t** options, will exit 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 will not be displayed in the inventory if the corresponding driver is not configured into IRIX.

SEE ALSO

lboot(1m), getinvent(3)

NAME

hosts – host name-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 host name
- 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 which 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 may 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 host names may 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

gethostbyname(3N), ifconfig(1M), named(1M), resolver(4), sys_id(4), hostname(5)

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's usually not used.

`5` Stop the UNIX system and enter firmware mode.

`6` Stop the UNIX 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, which may be defined to run certain commands, but which 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 will be killed. Other processes not started directly by **init** (such as **cron**) will 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** 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 a UNIX 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** will usually use 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 **/sbin/sulogin** cannot be found, then **init** will attempt 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** will only re-initialize 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 will 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 power-down 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/ioctl.syscon
/dev/console

SEE ALSO

login(1), **sh**(1), **stty**(1), **who**(1), **getty**(1M), **killall**(1M), **powerdown**(1M), **sulogin**(1M), **shutdown**(1M), **kill**(2), **inittab**(4), **system**(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 will assume that there is an error in the command string in the entry, and generate an error message on the system console. It will then refuse to respawn this entry until either 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 will loop; that is, it will go to firmware and reboot continuously.

If the **utmp** file cannot be created when booting the system, the system will boot 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.

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 may 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 will be processed. When *init* is requested to change *run-levels,* all processes which do not have an entry in the *rstate* field for the target *run-level* will be 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, then the process is assumed to be valid at all *run-levels.* There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which 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 actions 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* will cause *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*, where the process is still running from a previous *run-level* change, the program will not be 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, not wait for its termination; and when it dies, 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 will run 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* will enter *run-level* **6**. Additionally, if *init* does not find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.
sysinit	Entries of this type are executed before *init* tries to access the console (i.e., before the **Console Login:** prompt). It is expected that this entry will be only used 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

sh(1), who(1), getty(1M), init(1M), 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 (e.g., 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 file system 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, etc), 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 (i.e., 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 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), fs(4), efs(4), types(5).

NAME

inst – software installation tool

SYNOPSIS

inst [–**anAMNQXY**] [–**f** *source*] [–**m** *hardware*=*value*] [–**r** *target*] [–**u** *action*] [–**F** *selections-file*] [–**I** *product*] [–**R** *product*] [–**K** *product*] [**-P** 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** file systems 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 *IRIS Software Installation 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 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) manual 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 *IRIS Software Installation 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 will 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 will appear and the default location of the software distribution will be used unless the –**f** option is given. The software that will be installed will be selected by *inst* using an algorithm described in the *IRIS Software Installation Guide*.

NAME

killall – kill named processes

SYNOPSIS

/sbin/killall [[–] signal]
/sbin/killall [–gv] [–k secs] [[–]signal] [pname ...]
/sbin/killall [–gv] [–k secs] [–signame] [pname ...]
/sbin/killall –l

DESCRIPTION

killall sends a signal to a set of processes specified either 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 **/etc/shutdown.**

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 super-user.

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 matchs the real, saved, or effective user ID of the receiving process. The signal number can be preceded by a hyphen ("–"), which is required if *pname* looks like a signal number. A mnemonic name for the signal can be used; *killall* –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. If no signal value is specified, a default of **9** (KILL) is used.

The –**v** option reports if the signal was successfully sent. The –**g** option 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. The –**k** option 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* will wait 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, then the process is sent SIGKILL.

This command can be quite useful for killing a process without knowing its process ID. *Killall* can be used to stop a run-away 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

kill(1), ps(1), fuser(1M), shutdown(1M), signal(2)

NAME

lboot – configure bootable kernel

SYNOPSIS

/usr/sbin/lboot [–**m** master] [–**s** system] [–**b** boot] [–**n** mtune] [–**c** stune] [–**u** unix]

DESCRIPTION

The *lboot* command is used to configure a bootable UNIX kernel. Master files in the directory *master* contain configuration information used by *lboot* when creating a kernel. 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 will be included in the bootable kernel. For all included modules, *lboot* searches the *boot* directory for an object file with the same name as the file in *master*, but with a ".o" or ".a" appended. If found, this object is included when building the bootable kernel.

For every module in the *system* file specified via "VECTOR:", *lboot* takes actions to determine if a hardware device corresponding to the specified module exists. Generally, the action is a memory read at a specified base, of the specified size. If the read succeeds, the device is assumed to exist, and its module will also be 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*	This option specifies the directory containing the master files to be used for the bootable kernel. The default *master* directory is **$ROOT/var/sysgen/master.d**.
–**s** *system*	This option specifies the directory containing the system files. The default *system* directory is **$ROOT/var/sysgen/system**.
–**b** *boot*	This option specifies the directory where object files are to be found. The default *boot* directory is **$ROOT/var/sysgen/boot**.
–**n** *mtune*	This option specifies the directory where tunable parameters are to be found. The default *mtune* directory is **$ROOT/var/sysgen/mtune**.
–**c** *stune*	This option specifies the name of the file defining customized tunable parameter values. The default *stune* file is **$ROOT/var/sysgen/stune**.

−**r** *ROOT*	If this option is specified, **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**	This option makes *lboot* slightly more verbose.
−**u** *unix*	This option specifies the name of the target kernel. By default, it is **unix.new ,** unless the −**t** option is used, in which case the default is **unix.install**.
−**d**	This option displays debugging information about the devices and modules put in the kernel.
−**a**	This option is used to autoregister all dynamically loadable loadable kernel modules that contain a ′d′ and an ′R′ in their master files.
−**l**	This option is used to ignore the ′d′ in all master files and link all necessary modules into the kernel.
−**w**	This option is used to specify a "work directory" into which the master.c and edt.list files will be written. By default these files are written into the boot directory.
−**t**	This option tests if the existing kernel is up-to-date. If the kernel is not up-to-date, it prompts you to proceed. It compares the modification dates of the *system* 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 ‘‘.install’’ to the target name.
−**T**	This option performs the same function as the **-t** option, but does not prompt you to proceed.
−**L** *master*	This option 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 **$ROOT/var/sysgen/master.d** directory.
−**R** *master*	This option specifies the name of the dynamically loadable kernel module to register. *master* is the name of a master file in the **$ROOT/var/sysgen/master.d** directory.
−**U** *id*	This option is used to unload a dynamically loadable kernel module. *id* is found by using the lboot -V command.
−**W** *id*	This option is used to unregister a dynamically loadable kernel module. *id* is found by using the lboot -V command.
−**V**	This option is 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

This will read 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), system(4), mtune(4), stune(4), mload(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 will be 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 **cntrl-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 will 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 will be prompted for one at login time.
MANDPASS=NO	Like PASSREQ, but won’t allow users with no password to log in.
ALTSHELL=YES	If YES, the environment variable SHELL will be initialized.
UMASK=022	Default umask, in octal.
TIMEOUT=60	Exit login after this many seconds of inactivity (maximum 900, or 15 min.)
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 will be 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 non-negative, specify a grace period during which users with expired passwords will be 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 zero, 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*(3) and *syslogd*(1M) for details). No messages are sent to syslog if not set. Note 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*(3)).
SVR4_SIGNALS=YES	Use the SVR4 semantics for the SIGXCPU and SIGXFSZ signals. If SVR4_SIGNALS=YES, then the SVR4 semantics are preserved and all processes will ignore SIGXCPU and SIGXFSZ by default. If SVR4_SIGNALS=NO, then these two signals will 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 will cause behavior which varies depending on the login shell. This setting has no affect on processes which 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 may 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 non-zero, 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".

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 will be:

```
    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 may 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 will log 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 will be 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 may be logged in **/var/adm/loginlog** (if it exists) and the TTY line will be 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 path name (e.g., –**sh**). If this field in the password file is empty, then the default command interpreter, **/usr/bin/sh** is used. If this field is *, then the named directory becomes the root directory, the starting point for path searches for path names 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 will allow **login** to execute correctly, but you will also need to include additional files, like shells or applications, which the user is allowed to execute. Since these applications may in turn rely on additional shared libraries, it may also be necessary to place additional shared objects in **/usr/lib**. See the **ftpd(1)** man 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 may be expanded or modified by supplying additional arguments when **login** prints the prompt requesting the user's login name. The arguments may 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**

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 (e.g., **/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 will get, 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**) will be used. (If both are present the last one in file will be used.) If there is no such entry, then no dialup password will be required. In other words, the **/etc/d_passwd** entry for **/sbin/sh** is the default.

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 which is only readable by the superuser. Refer to the **pwconv**(1M) manual entry 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
/var/mail/*your_name*	mailbox for user *your_name*
/usr/lib/locale/*locale***/LC_MESSAGES/uxcore**	language-specific message file [See **LANG** on **environ**(5).]

SEE ALSO

mail(1), **newgrp**(1M), **pwconv**(1M), **sh**(1), **su**(1M).
loginlog(4), **passwd**(4), **profile**(4), **shadow**(4), **environ**(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.

NAME

lvck – check and restore consistency of logical volumes

SYNOPSIS

/sbin/lvck [-l lvtabname] [lvx]
/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 may 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 may 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 file name 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 which is a member of a logical volume is connected with the wrong id, the volume cannot be used since the device will not be correctly located from the labels. *Lvck* will print 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 will 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 may 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 will 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(7M), 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 will be 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

/sbin/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, since volumes may have failed to initialize. Failure to initialize could be due to missing physical disks, for example.

The information printed may be limited to specified logical volume devices by using options. These may 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(7)* device driver, which does not use this information.

DIAGNOSTICS

Invoked with no arguments, *lvinfo* simply prints the active volumes and is silent about any which are not initialized.
To obtain a printout of all logical volume devices whether initialized or not, invoke as:

```
/sbin/lvinfo /dev/rdsk/lv*
```

Invoked with arguments, it will print 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."
This message means that there is a special device file for a logical volume of that name, but it is not currently active. This may 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/lvx"
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

lvinit(1M), mklv(1M), lvck(1M), lvtab(4), lv(7M).

NAME

lvinit – initialize logical volume devices

SYNOPSIS

/sbin/lvinit [-l lvtabname] [volume_device_names]

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*, since 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.

Note: this implies that the root file system of a machine must reside on a regular partition rather than a logical volume, since 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 *lvn*, 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* man page for possible error messages.

SEE ALSO

mklv(1M), lvck(1M), lvtab(4), lv(7M).

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 will be used by the machine. 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 may be continued by the usual backslash convention as illustrated above. A '#' as the first non-white character indicates a comment; blank lines may be present in the file and will be 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 will access the logical volume. In the above example, the entry *lv0* implies that the logical volume will be 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 may 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 may 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, ie. it must not be one of the dedicated partitions such as the disk volume label, track replacement area etc.

Note that 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*(3), 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

lvinit(1M), mklv(1M), lvck(1M), getlvent(3), 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** command. Its arguments can be either targets in the file or assignments overriding parameters defined in the file. The targets .I 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 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 ptys by using the parameter override *MAXPTY=100*, or any other number between 1 and 199. See *pty*(7M) for details.
ips	Creates special files for ESDI disks connected to an Interphase ESDI disk controller. See *ips*(7M) for details.
ipi	Creates special files for IPI disks connected to a XYLOGICS IPI disk controller. See *ipi*(7M) for details.
dks	Creates special files for SCSI disks. See *dksc*(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.
xyl	Creates special files for SMD disks connected to a Xylogics SMD disk controller. See *xyl*(7M) for details.
qictape	Creates special files for 1/4-inch cartridge tape drives connected to an ISI QIC-O2 tape controller. See *ts*(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.
magtape	Creates special files for 1/2-inch tape drives connected to a Xylogics Model 772 tape controller. See *xmt*(7M) for details.
ikon	Creates special files for hardcopy devices connected to an Ikon 10088 (or 10088A) printer controller. This includes Versatec TTL and Versatec differential compatible devices, as well as (Tektronix-compatible) Centronics printers. See *ik*(7) for details.
gpib	Creates special files for the National Instruments GPIB-1014 controller and associated devices. See *gpib*(7) 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*(7) 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 ips ipi* and *xyl* drives, and then creates links by which one 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 and xmt tape devices, then makes links to *tape*, *nrtape*, *tapens*, and *nrtapens* for the first tape drive found, if one exists. It first checks for xmt, then for SCSI in descending target ID order.

mindevs	This option is shorthand for creating the *generic*, *links*, *pty*, *ttys*, *gro*, and *grin* 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

mknod(1M) make(1) install(1)

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 may 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 will be referred to as the *master* file, as though it was a single file. This will allow 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 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 left most 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	file system
s	software driver
f	STREAMS driver
m	STREAMS module
x	not a driver; a loadable module
k	kernel module
u	a stubs module which will be 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 chars. maximum)
Field 3:	software driver external major number; "–" 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; "–" 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 may 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 UNIX system kernel or other dependent module contains external references to a module, but the module is not configured, then these external references would be 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 may 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 in–line 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	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	number of devices per controller taken directly from the current master file entry

EXAMPLES

A sample *master* file for a shared memory module would be 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 will cause routines named *shmsys*, *shmexec*, etc., 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* will be allocated, and the structure *shminfo* will be allocated and initialized as specified.

A sample *master* file for a VME disk driver would be 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 will cause 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 will be generated, also, by the boot program. The declared arrays will all be 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

system(4), lboot(1M), mload(4)

NAME

mkboottape – make a boot tape

SYNOPSIS

/etc/mkboottape [–**f** output_file_name] [–**l** |**-x** [file ...] | file ...]

DESCRIPTION

mkboottape Is used to build, list, or extract boot tapes. A boot tape consists of a special directory which contains the list of file names, sizes, and offsets, and from one to 20 files. Filenames may be up to 16 characters in length.

The following options are understood:

-l	list the contents of the boot tape. In this case the file arguments are ignored.
-f *[file]*	specify an alternate output file or device. The default is */dev/tape*.
-x *[file ...]*	extract files from the boot tape. If no file names are given, all files are extracted. Extracted files have their original size (they are not null padded to a block multiple).

The typical use for this program is to create an output file which is copied to a boot tape with the *distcp*(1m) program, along with product images. The prom monitor understands the boot tape format, and can boot files from tapes created directly or indirectly via *mkboottape*.

NOTES

Unless you have means to create a stand alone program this utility is useful only listing or extracting the files. In some cases it may be used to extract the standalone programs into a directory, since most standalone programs can be booted from the filesystem via *sash* or over a network.

The directory is written with all numeric values in big-endian order, regardless of whether the CPU is big- or littlen-endian. mkboottape detects this and handles any required byte swapping when the **-x** or **-l** options are used. No byte swapping of filenames or files is performed by mkboottape.

SEE ALSO

distcp(1m), inst(1m)

FILES

/dev/tape - default output device

NAME

mkfs – construct a file system

SYNOPSIS

/sbin/mkfs [-q] [-a] [-i] [-r] [-n inodes] special [proto]
/sbin/mkfs [-q] [-i] [-r] special blocks inodes heads sectors cgsize cgalign ialign [proto]

DESCRIPTION

mkfs constructs a file system by writing on the *special* file using the values found in the remaining arguments of the command line. Normally *mkfs* prints the parameters of the file system to be constructed; the *-q* flag suppresses this.
If the *-i* flag is given, *mkfs* will ask for confirmation after displaying the parameters of the file system to be constructed.

The *-r* flag causes *mkfs* to write only the superblock, without touching other areas of the file system. See the section below on the recovery option.

The *-a* flag causes *mkfs* 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 file system, 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 see no benefit from this option.

When the first form of *mkfs* is used, *mkfs* obtains information about the device size and geometry by means of appropriate IOCTLs, and assigns values to the file system 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* is used, then all the file system parameters must be specified from the command line. Each argument other than *special* and *proto* is interpreted as a decimal number.

The file system parameters are as follows:

blocks is the number of *physical* (512 byte) disk blocks the file system will occupy.
Note that the current maximum limit on the size of an EFS filesystem is 16777214 blocks (two to the 24th power). This could also be expressed as 8 gigabytes. *mkfs* will not attempt to make a filesystem larger than this limit.
inodes is the number of inodes the file system should have as a minimum.
heads is an unused parameter, retained only for backward compatibility.
sectors is the number of sectors per track of the physical medium.
cgsize is the size of each cylinder group, in disk blocks, approximately.
cgalign is the boundary, in disk blocks, that a cylinder group should be aligned to.
ialign is the boundary, in disk blocks, that each cylinder group's inode list should be aligned to.

Once *mkfs* has the file system parameters it needs, it then builds a file system containing two directories. The file system'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* uses it as a prototype file and will take 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 bootfile name). It is present for backward compatibility; boot blocks are not used on SGI machines, and *mkfs* 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* about files and directories to be included in this file system.

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 **$** on line 7 tells *mkfs* to end the branch of the file system it is on, and continue from the next higher directory. It must be the last character on a line. The **:** on line 12 introduces a comment; all characters up until the following newline are ignored. Note that this means you may not have files in a prototype file whose name contains a **:**. The **$** 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 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 set-user-ID mode or not. The third is **g** or – for the set-group-ID mode. The rest of the mode is a 3-digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod*(1)).

Two decimal number tokens come after the mode; they specify the user and group IDs of the owner of the file.

If the file is a regular file, the next token of the specification may be a path name whence the contents and size are copied. If the file is a block or character special file, two decimal numbers follow which give the major and minor device numbers. If the file is a symbolic link, the next token of the specification is used as the contents of the link. If the file is a directory, *mkfs* 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* to write only the superblock, without touching the remainder of the file system space. This allows a last-ditch recovery attempt on a file system whose superblocks have been destroyed: by running *mkfs* 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 file system.

Note that this procedure will only be of use if the regenerated superblock matches the parameters of the original file system. If the file system was created using the long form invocation, parameters identical to the original invocation must be given with the *-r* option. Note also that file system 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 file systems created under IRIX versions other than the version being run.

It should be clear that this is a limited recovery facility; it will not help if, for example, the root directory of the file system has been destroyed.

SEE ALSO

chmod(1), mkfp(1M), dir(4), fs(4)

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 which 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 *lvn* 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.

Note that an existing logical volume may 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 which 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* will unconditionally refuse 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 *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 will be erased. This avoids errors resulting from erroneous attempts to mount individual disks which 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* and no message or filesystem modification then 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 which is inconsistent with the volume specified in the lvtab entry. (Note: 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 may 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 will be a logical volume label file named '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, will remove certain links and replace 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

lvinit(1M), lvck(1M), lvtab(4), lv(7M), growfs(1M).

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 or a library module.

Module Configuration

Each loadable module should contain the string:

char ***prefix**mversion = 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:

-non_shared -coff -G0 -Wc,-pic0 -r -d -c -jalr

-non_shared	Produce a static executable. The output object created will not use any shared objects during execution.
-coff	Produce a COFF object.
-G0	Disable global pointer since it is not supported for loadable modules.
-Wc,-pic0	Do not allocate extra stack space which is not necessary for non_shared coff 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.

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:

-non_shared -coff -G0 -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.

For more detailed information, see the **lboot**(1M), **ml**(1M) and **master**(4) manual entries.

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 5 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) manual entry. 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) manual entries.

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 Programming 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) manual entries.

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 COFF object files are supported as loadable library modules.

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.

Runtime Symbol Table

A runtime symbol table which contains kernel routines and global data that modules can link against is created by lboot from the tables in **master.d/rtsymtab** and **master.d/*.exports**. Only routines and globals that are always present in the kernel should be added to **master.d/rtsymtab**.

If a routine or global exists in a module that could be configured out of the kernel, it should be added to an xxx.exports file which will not be included in the kernel if the module is configured out. See master.d/idev.exports, for example.

If a loadable module contains globals in its master.d file, an xxx.exports file will need to be created, which includes those globals, so that they will be added to the runtime symbol table.

For more information on the runtime symbol table see **master.d/rtsymtab**.

Debugging Loadable Modules

Symmon 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) manual 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.

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.

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**(1M) 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:

```
cc -non_shared -coff -G0 -r -d -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

lboot(1M), ml(1M), master(4), cc(1), ld(1), mtune(4), systune(1M), symmon(1M) and the IRIX Device Driver Programmer's Guide.

NAME

mount, umount – mount and dismount file systems

SYNOPSIS

/sbin/mount
/sbin/mount [–**M** *altmtab*] [–**P** *prefix*] –**p**
/sbin/mount [–**h** *host*] [–**fnrv**]
/sbin/mount –**a**[**cfnv**] [–**t** *type*]
/sbin/mount [–**cfnv**] [–**t** *type*] [–**b** *list*]
/sbin/mount [–**cfnrv**] [–**t** *type*] [–**o** *options*] *fsname dir*
/sbin/mount [–**cfnrv**] [–**o** *options*] *fsname* | *dir*

/sbin/umount –**a**[**kv**] [–**t** *type*]
/sbin/umount –**h** *host* [–**kv**] [–**b** *list*]
/sbin/umount [–**kv**] *fsname* | *dir*

DESCRIPTION

Mount attaches a named file system *fsname* to the file system 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 file system is unmounted. If *fsname* is of the form host:path, the file system type is assumed to be **nfs**.

Umount unmounts a currently mounted file system, which can be specified either as a mounted-on *directory* or a *file*system.

Mount and **umount** maintain a table of mounted file systems 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 file systems described in */etc/fstab*. (In this case, *fsname* and *dir* are taken from */etc/fstab*.) If a type is specified, all of the file systems in */etc/fstab* with that type are mounted. Filesystems are not necessarily mounted in the order listed in */etc/fstab*.

–**b** *list* ‘‘all-but’’ – attempt to mount all of the file systems 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 file system being mounted, and if it indicates that the file system is dirty, call *fsck*(1M) to clean the file system. *fsck* is passed the –**D** and –**y** options.

−f Fake a new */etc/mtab* entry, but do not actually mount any file systems.

−h *host* Mount all file systems listed in */etc/fstab* that are remote-mounted from *host*.

−n Mount the file system 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 file systems in a format suitable for use in */etc/fstab.*

−r Mount the specified file system read-only. This is a shorthand for:

mount −o ro *fsname dir*

Physically write-protected and magnetic tape file systems 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 file system type. The accepted types are **proc**, **efs**, **nfs**, **fd**, **cachefs**, **dos**, **hfs** and **iso9660**; see *fstab*(4) for a description of these file system 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.

−v Verbose − **mount** displays a message indicating the file system 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 file systems 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 file systems currently mounted except for those associated with the directories contained in *list*. *list* consists of one or more directory names separated by commas.

−h *host* Unmount all file systems 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 file systems of a given file system type. The accepted types are **proc**, **efs**, **nfs**, **fd**, **dos**, **hfs**, and **iso9660**.

–**v** Verbose – **umount** displays a message indicating the file system being unmounted and any problems encountered.

EXAMPLES

```
mount /dev/usr /usr                mount a local disk
mount –avt efs                     mount all efs file systems; be verbose
mount –t nfs server:/d /net/d      mount remote file system
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 file system
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 file systems except /foo
```

FILES

/etc/fstab	file system table
/etc/mtab	mount table

SEE ALSO

mount(2), umount(2), fstab(4), mtab(4)
mountd(1M), nfsd(1M) if NFS is installed.
fsck(1M)

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 file system is to be mounted is a symbolic link, the file system 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 eoe2.sw.cdrom. This package must be installed in order to mount iso9660 filesystems.

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.

Parameters in the *mtune* directory are grouped together in *modules*, 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* module. A parameter's tuning information is stored in the *mtune* directory in a file with the same name as the module. The syntax of an *mtune* module file is given below:

```
<module name>: <flag>
<parameter name 1>   <default value> <minimum value> <maximum value>
<parameter name 2>   <default value> <minimum value> <maximum value>
```

Lines that end with a colon character '':'' are module names. Parameters are grouped together in modules so that one ''sanity checking'' function can be used to verify the values and the dependencies between these variables. The module name is optional if there is only one group in the module. For this case, the configuration tools will use the module name as the group name. The group name is then followed by a flag. The flag can be either ''run'' or ''static''. If the flag is *run*, that means 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 only be changed by creating a new kernel and booting that kernel.

Each tunable parameter is specified by a single line in the file. Blank lines and lines beginning with "#" or "*" are considered comments and are ignored. The syntax for each line is:

```
<parameter name>   <default value> <minimum value> <maximum value>
```

parameter name:	This is 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.
default value:	This is 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.
minimum value:	This is 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.

maximum value:	This is 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.

FILES

/var/sysgen/mtune/*	default system tunable parameters
/var/sysgen/stune	local settings for system tunable parameters

SEE ALSO

stune(4)
lboot(1M), systune(1M)

NAME

nvram, sgikopt – get or set non-volatile RAM variable(s)

SYNOPSIS

nvram [–**v**] [*name* [*value*]]
sgikopt [*name*...]

DESCRIPTION

Nvram may be used to set or print the values of non-volatile RAM variables. If *name* is specified, *nvram* prints the corresponding value. If *value* is specified and *name* is defined in non-volatile RAM, *nvram* replaces *name*'s definition string with *value*. The –**v** option causes *nvram* to print a line of the form *name=value* after getting or setting the named variable. When invoked with no arguments, all known variables are displayed in the *name=value* form.

If invoked as *sgikopt*, more than one name may be given. Names that do not match known variables are ignored. The exit status is 1 if any arguments don't match, and 0 otherwise.

NOTES

Non-volatile RAM contains a small set of well-known strings at fixed offsets. *Nvram* may not be used to define new variables.

Only the super-user may set variables.

The term "Non-volatile RAM" is somewhat misleading, because some variables are placed only in volatile RAM, and will be reset on power-up. Different models 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 machines support the ability to change the contents of non-volatile memory with the *nvram* command. To change the contents of non-volatile memory on those machines you must use the PROM monitor *setenv* command.

SEE ALSO

sgikopt(2), syssgi(2), prom(1m)

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 new-line. 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 may be safely ignored for users local to IRIX systems. The *pwck*(1m) command checks for the greatest possible portability on names, and will complain about user names that will 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, then 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

A & 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 may be used to implement secure or restricted logins, in a manner similar to *ftp*(1c).

Password aging is effected 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 will be forced to change his password. The next character, *m* say, denotes the minimum period in weeks which must expire before the password may be changed. If the second character is omitted, zero week is the default minimum. *M* and *m* have numerical values in the range 0–63 that correspond to the 64-character alphabet shown above (i.e., **/** = 1 week; **z** = 63 weeks). If *m* = *M* = 0 (derived from the string **.** or **..**) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If *m* > *M* (signified, e.g., by the string **./**) only the super-user will be 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 on a 'wire' your ID to remote systems. See system calls like chmod(2).

UID –2 NFS ’nobody’. Note that because uid_t is unsigned, -2 is mapped to UIDMAX+1 or 60001.

UID 60002 to 65535
Invalid.

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 60000 normal users (start at 100).

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* will have 63 weeks of maximum password aging and 1 week of minimum password aging. Programs that use the GECOS field will replace the & with ‘Bill’. The user *john* will have his password entry in the NIS incorporated without change; anyone in the netgroup *documentation* will have their password field disabled, and anyone else will be 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).
getpwent(3), **crypt**(3), **a64l**(3C).
group(4), **netgroup**(4), **exports**(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*(1) 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 ˆc
# 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), tset(1), tput(1), su(1M), 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

prom – PROM monitor

DESCRIPTION

The PROM monitor is a program that resides in permanently programmed read-only memory, which controls the startup of the machine. 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 machine to machine. Description of various commands, options and interfaces below may not apply to the PROM in your machine, and may vary between machines. Furthermore, since PROMs are not normally changed after the manufacture of the system, newly added features will not be present older machines.

Some machines, 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 will display a menu of alternate boot up options. These other choices allow various types of system maintenance to be performed and are described below:

1. Start System

This option will cause 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 will first attempt to find a tape drive on the system and if one is found, it will prompt the user to insert the installation tape in it. If one is not found, then installation is expected to take place by Ethernet. In this case, the PROM will prompt the user for the name of the machine that will be used as the server.

Installation 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 will perform a thorough test of the CPU board and any graphics boards present. It will then report a summary.

4. Recover System

This option may be used to perform special system administration tasks such as restoring a system disk from backup tapes. It follows a similar sequence for installing system software, but instead of starting the installation program, it invokes an interactive restoration tool.

5. Enter Command Monitor

Additional functions may 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 non-volatile RAM, which means that their value will be preserved even after the power to the machine 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 file name with a device name as shown:

devicename(*controller*,*unit*,*partition*)*filename*

Valid device names include:

tpsc	SCSI tape drive
dksc	SCSI disk drive
dkip	ESDI disk drive
ipi	IPI disk drive
xyl	SMD disk drive
tpqic	VME-based Quarter-Inch tape drive
bootp	Ethernet by BOOTP and TFTP protocols

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 machine to machine.

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 will only list those devices ''known'' to the PROM and may not include all optional boards.

ls *device*
List files contained on the device specified. This may 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 file systems.

off Turns off the power. Only supported on a subset of machines with software power control.

passwd
Set the PROM password. The PROM password may be set to restrict operation of certain PROM modes. With a password set, any attempt to do anything other than a standard system boot will require that the password be re-entered. The password is remembered after the system is powered off.

If the password is forgotten, some machines allow the super-user 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 which 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 to 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 non-volatile 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 will function without restriction.

setenv [-p] *variable value*
Set the specified environment variable to a particular value. Environment variables that are stored in non-volatile 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 non-volatile RAM. This is particularly useful for setting frequently used options when starting up the system. Note that a fixed non-volatile RAM variable will not be superseded by this option, but the command will behave as if the **-p** flag was 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 may be entered directly, which the PROM will attempt to load and execute.

PROM Environment Variables

netaddr
Used when booting or installing software from a remote machine by Ethernet. This variable should be set to contain the Internet address of the machine. This variable is stored in non-volatile RAM.

dbaud Diagnostic baud rate. The variable 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 non-volatile 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 file name 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 file system 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 file system and swap area are determined by reading the volume header. See *vh*(7M) for more information. This variable is stored in non-volatile 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 will display the menu and wait 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 non-volatile RAM.

boottune

Selects among the availiable boot tunes, and 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 non-volatile RAM.

autopower

On machines with software power control a setting of ''y'' will allow 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 non-volatile RAM.

console

The system console may 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'', then the console is assumed to be a terminal connected to the first serial port. In addition, som systems also accept ''d2'' for a terminal connected to second serial port. Lastly, this can be over-ridden on some systems by removing the password jumper and the console will be forced to ''g'', which is useful for for recovering from setting the console to ''d'' when a terminal is not available. This variable is stored in non-volatile RAM.

diskless

If set to ''1'', the kernel will assume that the machine is to be started up as a diskless node. This variable is stored in non-volatile 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 non-volatile RAM.

nogfxkbd

If set to ''1'', the system will 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 will switch to the serial terminal console and wait for a command. This variable is stored in non-volatile RAM.

notape If set to ''1'', the PROM will assume that the Ethernet is to be used for software installation or system recovery even if a tape drive is present on the machine. 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 tone generated on machines with integral audio hardware. This variable is stored in non-volatile RAM.

pagecolor
Sets the background color of the textport set with a six character string of hex rgb values. This variable is stored in non-volatile 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 will return 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 non-volatile 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 non-volatile 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 to clean up various loop-holes 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 manual page.

ARCS pathnames are tied directly to the hardware inventory, which is stored in a tree that represents the machines 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 will match the first device described by the pathname, so full pathnames are not always required. The ‘‘-p’’ option to hinv will print the pathnames to all user accessible devices. Some examples of common pathnames:

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
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 non-volatile RAM, but is normally left unset, which allows the PROM will automatically configure it at system power-on.

OSLoader
The operating system loader. For IRIX, this is sash. This variable is stored in non-volatile RAM, but is normally left unset, which allows the PROM will automatically configure it at system power-on.

SystemPartition
Device where the operating system loader is found. This variable is stored in non-volatile RAM, but is normally left unset, which allows the PROM will automatically configure it at system power-on.

OSLoadFilename
The file name of the operating system kernel. For IRIX this is /unix. This variable is stored in non-volatile RAM, but is normally left unset, which allows the PROM will automatically configure it at system power-on.

OSLoadOptions
The contents of this variable are appended to the boot command constructed when auto-booting the system. This variable is stored in non-volatile 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 non-volatile 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 will work.

For example the traditional 1-5 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 1-5 key.

The only major user interface changes is 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 machine. 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 will automatically configure the keyboard to use its native key map. To override the default this should be set to a three to five character string. The following strings may be 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 on can select between swiss french and swiss german by setting **keybd** to ‘‘d’’ or ‘‘D’’ for the german map. On machines with PC keyboards, a string not matching one of the above will be passed to the X server and used as the name of the keyboard map to load. This variable is stored in non-volatile RAM.

diagmode
: If set to ‘‘v’’ then power-on diagnostics are allowed to be 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 non-volatile 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 machine 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 seven 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. Re-initialize the PROM with the **init** command:

   ```
   init
   ```

NAME

prtvtoc – print disk volume header information.

SYNOPSIS

/etc/prtvtoc [header_device_name] device

DESCRIPTION

prtvtoc prints a summary of the information in the volume header of a disk. (See **vh**(7m)). The command can normally be used only by the superuser.

The *device* 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 may be omitted.

If no name is given, the information for the root disk is printed.

prtvtoc prints information about the disk geometry (number of cylinders, heads, etc.), followed by information about the partitions. For each partition, the type is indicated (for example, file system, raw data and so forth). Cylinders may be non-integral values, as they may not correspond to actual physical values, for some drive types. For file system partitions *prtvtoc* shows if there is actually a filesystem on the partition, and if it is mounted, the mount point is shown.

The following options to **prtvtoc** may 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**.
–**m***mnttab*	Use the file *mnttab* instead of **/etc/mount**.

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         efs  yes      3321   (   3)      32103   (  29)   /
 1         raw          35424   (  32)      81918   (  74)
 6         efs  yes    117342   ( 106)    1959390   (1770)   /usr
 7         efs           3321   (   3)    2073411   (1873)
 8      volhdr              0   (   0)       3321   (   3)
10      volume              0   (   0)    2076732   (1876)
```

SEE ALSO

dvhtool(1M), **fx**(1m), **vh**(7m). **dks**(7m). **ips**(7m). **ipi**(7m). **usraid**(7m).

NAME

ps – report process status

SYNOPSIS

ps [options]

DESCRIPTION

ps prints certain 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 given in descending order according to volume and range of information provided:

–**e**	Print information about **e**very process now running.
–**d**	Print information about all processes except process group leaders.
–**a**	Print information about **a**ll processes most frequently requested: all those except process group leaders and processes not associated with a terminal.
–**j**	Print session ID and process group ID.
–**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.
–**f**	Generate a **f**ull listing. (See below for significance of columns in a full listing.)
–**l**	Generate a **l**ong listing. (See below.)
–**c**	Print information in a format that reflects the scheduler properties. The –**c** options affects the output of the –**f** and –**l** options, as described below.
–**n** *name*	This argument is obsolete and is no longer used.
–**t** *termlist*	List only process data associated with the terminal given in *termlist*. Terminal identifiers consist of the device’s name (e.g., **ttyd1**, **ttyq1**).
–**p** *proclist*	List only process data whose process ID numbers are given in *proclist*.
–**u** *uidlist*	List only process data whose user ID number or login name is given in *uidlist*. In the listing, the numerical user ID will be printed unless you give the –**f** option, which prints the login name.
–**g** *grplist*	List only process data whose process group leader’s ID number(s) appears 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.)
–**s***sesslist*	List information on all session leaders whose IDs appear in *sesslist*.

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; **all** means that the heading always appears. Note that these two options determine only what information is provided for a process; they do not determine which processes will be listed.

F (l) Flags (hexadecimal and additive) associated with the process:

01	Process is a system (resident) process.
02	Process is being traced.
04	Stopped process has been given to parent via *wait*(2).
08	Process is sleeping at a non-interruptible priority.
10	Process is in core.
20	Process user area is in core.
40	Process has enabled atomic operator emulation.
80	Process in stream poll or select.

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.
C	(f,l)	Processor utilization for scheduling. Not printed when the −**c** option is used.
CLS	(f,l)	Scheduling class. Printed only when the −**c** option is used. The values printed for **CLS** are the two character mnemonics for the scheduler queues displayed by the −**q** option of *pset*(1M).
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. Only processes in the time-sharing class have a nice value. Processes in other scheduling classes will have their 2 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.
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. A page is 4096 bytes.
SID	(j)	Session ID. This can be used with the −**s** option.
PGID	(j)	Process group leader ID. This can be used with the −**g** option.
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 will each count all valid resident pages as part of their own resident size. A page is 4096 bytes.
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>**.

FILES

/dev	
/dev/tty*	
/etc/passwd	UID information supplier
/tmp/.ps_data	internal data structure. This file is used to improve the performance of *ps* by caching uid to username translations, kernel info, and some device information. It is re-created 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 re-created. Note that new NIS users may have jobs reported as numeric values, since the ps_data file won't be re-created automatically; removing this file and re-running *ps* will fix the problem.

SEE ALSO

getty(1M), gr_osview(1), gr_top(1), kill(1), nice(1), pset(1M), top(1).

WARNING

Things can change while *ps* is running; the snap-shot it gives is only true for a split-second, 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 will attempt to report on processes associated with the controlling terminal. In this situation, if *stdin*, *stdout*, and *stderr* are all redirected, *ps* will not find a controlling terminal, so there will be 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

/usr/sbin/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 lower-case 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 may be restored to its pre-conversion 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 will cause **pwconv** to be run, and therefore the backup files to be overwritten, each time an entry is added, deleted, or modified.

NOTES

1. There is no NIS equivalent to /etc/shadow. Therefore, if a user does not have an entry in /etc/shadow, they will not be able to 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 will still be available, just not passwords.

2. *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 [–**pv**] file1 file2
rcp [–**p**] [–**rv**] file ... directory

DESCRIPTION

Rcp copies files between machines. Each *file* or *directory* argument is either a remote file name of the form ''remhost:path'', or a local file name (containing no ':' characters, or a '/' before any ':'s). Hostnames may also take the form ''remuser@remhost'' to use the user name *remuser* rather than the current user name on the remote host.

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

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 –**p** option causes *rcp* to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the *umask*. The –**v** option causes the file name to be printed as it is copied to or from a remote host.

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

Rcp does not prompt for passwords; your current local user name must exist on *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), rsh(1C), rlogin(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* will 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*(3)-based programs.

Rcp cannot handle file names which have imbedded 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 tape(s) to disk. if no file or directory is specified *Restore* will copy all the files found on the tape to disk.

Files are restored into the current directory if the backup tape contains "." relative path names.

Files on disk are overwritten even if they are more recent than the respective files on tape.

If a tape drive attached to a remote host is used for restore, the name of the remote host needs to be specified with the –**h hostname** option on the command line. For remote restore to successfully work, the user should have a TCP/IP network connection to the remote host and also have "guest" login privileges on that host.

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.**

The *Restore* command expects the backup tape to be in the special "bru" 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 file system restore

SYNOPSIS

/sbin/restore key [name ...]
/sbin/rrestore key [name ...]

DESCRIPTION

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 file system or to restore an incremental dump tape after a full level zero 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, then the root directory is extracted, which results in the entire content of the tape being extracted, unless the **h** key has been specified.

t The names of the specified files are listed if they occur on the tape. If no file argument is given, then the root directory is listed, which results in the entire content of the tape being listed, unless the **h** key has been specified. Note that the **t** key replaces the function of the old *dumpdir* program.

i This mode allows interactive restoration of files from a dump tape. After reading in the directory information from the tape, **restore** provides a shell like interface that allows the user to move around the directory tree selecting files to be extracted. The available commands are given below; for those commands that require an argument, the default is the current directory.

ls [arg] – List the current or specified directory. Entries that are directories are appended with a ''/''. Entries that have been marked for extraction are prepended with a ''*''. If the verbose key is set the inode number of each entry is also listed.

cd arg – Change the current working directory to the specified argument.

pwd – Print the full pathname of the current working directory.

add [arg] – The current directory or specified argument is added to the list of files to be extracted. If a directory is specified, then it and all its descendents are added to the extraction list (unless the **h** key is specified on the command line). Files that are on the extraction list are prepended with a ''*'' when they are listed by **ls**.

delete [arg] – The current directory or specified argument is deleted from the list of files to be extracted. If a directory is specified, then it and all its descendents are deleted from the extraction list (unless the **h** key is specified on the command line). The most expedient way to extract most of the files from a directory is to add the directory to the extraction list and then delete those files that are not needed.

extract – All the files that are on the extraction list are extracted from the dump tape. **restore** will ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

setmodes – All the directories that have been added to the extraction list have their owner, modes, and times set; nothing is extracted from the tape. This is useful for cleaning up after a **restore** has been prematurely aborted.

verbose – The sense of the **v** key is toggled. When set, the verbose key causes the **ls** command to list the inode numbers of all entries. It also causes **restore** to print out information about each file as it is extracted.

help – List a summary of the available commands.

quit – **restore** immediately exits, even if the extraction list is not empty.

The following characters may be used in addition to the letter that selects the function desired.

b The next *argument* to **restore** 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 will only be 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 file system with the command

```
dump 0f - /usr | (cd /mnt; restore xf -)
```

If the name of the file is of the format *machine:device* then 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 file name *argument* is of the form *user@machine:device,* **restore** will attempt 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** will not ask whether it should abort the restore if gets a tape error. It will always try to skip over the bad tape block(s) and continue as best it can.

m **restore** will extract by inode numbers rather than by file name. This is useful if only a few files are being extracted, and one wants to avoid regenerating the complete pathname to the file.

h **restore** 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 which selects the dump file when there are multiple dump files on the same tape. File numbering starts at 1.

n Only those files which 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 super-user (or with the effective user id of zero). This is to provide Berkeley style semantics. This can be overridden with the **o** option which will result 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** will attempt to continue the restore.

If the dump extends over more than one tape, **restore** will ask the user to change tapes. If the **x** or **i** key has been specified, **restore** will also ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

There are numerous consistency checks that can be listed by **restore.** Most checks are self-explanatory or can ''never happen''. Common errors are given below.

Converting to new file system format.
: A dump tape created from the old file system has been loaded. It is automatically converted to the new file system format.

<filename>: not found on tape
: The specified file name was listed in the tape directory, but was not found on the tape. This is caused by tape read errors while looking for the file, and from using a dump tape created on an active file system.

expected next file <inumber>, got <inumber>
: A file that was not listed in the directory showed up. This can occur when using a dump tape created on an active file system.

Incremental tape too low
: When doing incremental restore, a tape that was written before the previous incremental tape, or that has too low an incremental level has been loaded.

Incremental tape too high
: When doing incremental restore, a tape that does not begin its coverage where the previous incremental tape left off, or that has too high an incremental level has been loaded.

Tape read error while restoring <filename>
Tape read error while skipping over inode <inumber>
Tape read error while trying to resynchronize
: A tape read error has occurred. If a file name is specified, then its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, then no extracted files have been corrupted, though files may not be found on the tape.

resync restore, skipped <num> blocks
: After a tape read error, **restore** 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 which 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 which has more space available.

EXAMPLES

```
restore r
```

will restore the entire tape into the current directory, reading from the default tape device */dev/tape.*

```
restore rf guest@kestrel.sgi.com:/dev/tape
```

will restore 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
```

will restore the three specified files into the current directory, reading from the default tape device */dev/tape.*

```
restore x /dev/dsk
```

will restore the entire /dev/dsk directory and subdirectories recursively into the current directory, reading from the default tape device */dev/tape*

```
restore rN
```

will read 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
```

will restore (recursively) all files in the given directory /usr/dir/foo. However, no existing files are overwritten.

```
restore xn /usr/dir/bar
```

will restore (recursively) all files which 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, then the file will be 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, then the file will be created in that directory.

./restoresymtable
Information is passed between incremental restores in this file.

SEE ALSO

dump(1M), mount(1M), mkfs(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 file systems.

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*] [–**8**] [–**L**]
rlogin username@rhost [–**e** *c*] [–**8**] [–**L**]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* 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 the *username@rhost* format.

Each host has a file */etc/hosts.equiv* which contains a list of *rhost*'s with which it shares account names. (The host names must be the standard names as described in *rsh*(1C).) When you *rlogin* as the same user on an equivalent host, you don't need to give a password. Each user may also have a private equivalence list in a file .rhosts in his login directory. Each line in this file should contain an *rhost* and a *username* separated by a space, giving additional cases where logins without passwords are to be permitted. If the originating user is not equivalent to the remote user, then a login and password will be prompted for on the remote machine 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 environment TERM 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 ˆS and ˆQ and flushing of input and output on interrupts are handled properly. The optional argument –**8** allows an eight-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 ˆS/ˆQ. The argument –**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 ''˜ˆZ'' (where ˆZ, control-Z, is the suspend character) will suspend the rlogin session if you are using *csh*(1). A different escape character may be specified by the –**e** option. There is no space separating this option flag and the argument character.

SEE ALSO

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

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 function is 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* 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 file system 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*(3B)). 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 will force *savecore* to attempt to save the core image regardless of the state of this magic number. This may be necessary since *savecore* will always clear the magic number after reading it. If a previous attempt to save the image failed in some manner, it will still be possible to restart the save with this option.

–v	Give more verbose output.

DIAGNOSTICS

"warning: /unix may not have created core file" is 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" is 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 will enable *savecore* to find name list information about the current state of the running system from the file */unix*.

FILES

/unix	current UNIX
/var/adm/crash	default place to create dump files
/var/adm/crash/bounds	number for next dump file
/var/adm/crash/minfree	minimum file system free space
/etc/config/savecore.options	site-specific command-line options

SEE ALSO

nlist(3X), uncompvm(1M)

NAME

setmnt – establish mount table

SYNOPSIS

/sbin/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 file system's *special file* (for example, **/dev/dsk/dks?d?s?**) and *dir* is the mount-point of that file system. 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 which 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 (i.e., 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 (non-zero) exit status. An arbitrary number of newlines may 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, then 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 file-name generation (see ''File Name 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, then 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, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

(*list***)**
: Execute *list* in a sub-shell.

{*list***;**}
: *list* is executed in the current (that is, parent) shell. The **{** must be followed by a space.

name **()** {*list***;**}
: Define a function which is referenced by *name.* The body of the function is the *list* of commands between { and }. The *list* may appear on the same line as the **{**. If it does, the { and *list* must be separated by a space. The } may not 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 new-line to be ignored.

Command Substitution

The shell reads commands from the string between two grave accents (**``**) and the standard output from these commands may be used as all or part of a word. Trailing new-lines 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 may 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 (\**"**) will be removed; otherwise, it will be left intact.

If a backslash is used to escape a new-line character (**\\new-line**), both the backslash and the new-line 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 \, **\`**, **"**, **new-line**, 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 may be assigned values by **set**. Keyword parameters (also known as variables) may 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 may not 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 sub-shells.

! 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 may not 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 will check 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 will check before each prompt.

MAILPATH
A colon (**:**) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by **%** and a message that will 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 **new-line**.

SHACCT
If this parameter is set to the name of a file writable by the user, the shell will write 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 may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a *simple-command* or may 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*, and
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) **\new-line** is ignored, and
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 which 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 (i.e., *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 will be 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.

File Name 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 file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character **.** at the start of a file name 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:

; & ( ) | ^ < > new-line space tab

A character may be *quoted* (i.e., 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 may 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 **\new-line** 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 may 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 file name 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 **\new-line** is removed before parameter and command substitution. If a backslash precedes characters other than \, `, ", **$**, and new-line, 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 new-line is typed and further input is needed to complete a command, the secondary prompt (i.e., 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 may 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* may be augmented by prefixing it with one or more assignments to parameters. Thus:

```
TERM=450 cmd                       and
(export TERM; TERM=450; cmd)
```

are equivalent (as far as the execution of *cmd* is concerned if *cmd* is not a Special Command). If *cmd* is a Special Command, then

```
TERM=45 cmd
```

will modify 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**, ... 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 path name, which 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 will not be 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 sub-shell is spawned to read it. A parenthesized command is also executed in a sub-shell.

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 path name, which 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 may not 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 may 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 will also cause 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 will be done are indicated by an asterisk (∗) adjacent to the *hits* information. *Cost* will be 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 super-user may raise the hard limits, but a user may 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 which 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 that will be created, *memoryuse*, the maximum amount of physical memory a process may 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, etc..

The *maximum-use* may 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’ may 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 may 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*, etc., with leftover words assigned to the last *name*. Lines can be continued using **\new-line**. Characters other than **new-line** 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 may not 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 which are modified or created for export.
–**e**	Exit immediately if a command exits with a non-zero exit status.
–**f**	Disable file name 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 may 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 may be read). If *n* is omitted, the current limit is printed. You may lower your own ulimit, but only a super-user (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 super-user may 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 will be 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. Non-interactive 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 non-current job.

The acceptable syntax for a Job Identifier is of the form:

`%`*jobid*

where, *jobid* may 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 will be reported. The following options will 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 file name part of the last entry in the */etc/passwd* file (see *passwd*(4)); (2) the environment variable **SHELL** exists and *rsh* is the file name part of its value; (3) the shell is invoked and *rsh* is the file name 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 (ie., **/usr/rbin** that can be safely invoked by a restricted shell. Some systems also provide a restricted editor, *red*.

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively 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 will be 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), dup(2), echo(1), env(1), exec(2), fork(2), getopts(1), getrlimit(2), intro(1), login(1), newgrp(1), pipe(2), profile(4), pwd(1), signal(2), test(1), ulimit(2), umask(1), wait(1)

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 ''File Name Generation,'' above). For example, **cat file1 >a*** will create a file named **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 will continue 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**(1).

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 new-line. 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 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 may 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), **`passwd`**(1), **`passmgmt`**(1M), **`pwconv`**(1M), **`passwd`**(4).
`getspent`(3C), **`putspent`**(3C).

NOTES

Shadow passwords may 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> = <value>
```

parameter name:	This is the name of the tunable parameter.
value:	This field contains the new value for the tunable parameter.

The file **stune** normally resides in **/var/sysgen**. You may 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

mtune(4)
lboot(1M), systune(1M)

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 which 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

network(1M), lockd(1M), statmon(4)

BUGS

The crash of a site is only detected upon its recovery.

NAME

su – become super-user or another user

SYNOPSIS

su [–] [name [arg ...]]

DESCRIPTION

su allows one to become another user without logging off. The default user *name* is **root** (i.e., super-user).

To use *su*, the appropriate password must be supplied (except as described below). If the password is correct, *su* will execute a new shell with the real and effective user ID set to that of the specified user. The new shell will be the optional program 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** (*cntrl-d*) to the new shell.

Su prompts for a password if the specified user's account has one. However, *su* will 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** will give 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 will be 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 first 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**. 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*(1), respectively. If the user's program is other than **/bin/sh**, then **.profile** is invoked with an *arg0* of *–program* by both *login*(1) and *su*(1).

All attempts to become another user using *su* are logged in the log file **/var/adm/sulog** by default.

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

SEE ALSO

env(1), login(1), sh(1), passwd(4), cshrc(4), profile(4), rhosts(4), environ(5)

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 machines.

To use symmon, several steps must be taken to prepare the machine. First, symmon is not installed on the system as shipped from the factory and must be manually installed by the user. This can be done by installing the ''Debugging Kernels'' subsystem in the development option tape. Second, 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. Third, the program **setsym** needs to be run on the newly generated kernel to allow symmon to recognize symbols in it. Finally, symmon needs to be installed in the volume header of the root drive with *dvhtool*(1M). This will normally happen 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 will automatically try to load symmon from the same source. Once symmon is loaded, the system will operate 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, a control-a is typed. 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 machine state. Not all of the commands listed below are supported on all machines. Some commands take memory addresses as arguments. Address may be given either 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 may be used to specify byte, halfword, or full word data and the **-o**, **-d**, **-x**, and **-c** flags may be used to specify octal, decimal, hexadecimal, or ASCII data formats.

The specified range of memory to dump may take the form: a) *base* for a single location, b) *base#count* for *count* locations starting at *base,* or c) *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 will enter a command repeat mode where a null command will cause 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** may 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 may by determined by listing the set breakpoints with the brk command.

wpt [r|w|rw] [0|phys addr]
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 will trip on any access within the next eight bytes. An argument of zero 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 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

syslogd – log systems messages

SYNOPSIS

/usr/etc/syslogd [–**f***configfile*] [–**m***markinterval*] [–**p***logpipe*] [–**d**]

DESCRIPTION

Syslogd reads and logs messages into a set of files described by the configuration file /etc/syslog.conf. Each message is one line. A message can contain a priority code, marked by a number in angle braces at the beginning of the line. Priorities are defined in <*sys/syslog.h*>. *Syslogd* reads from the stream device */dev/log*, from an Internet domain socket specified in */etc/services*, and from the special device */dev/klog* (to read kernel messages).

Syslogd reads its configuration when it starts up and whenever it receives a hangup signal. Lines in the configuration file have a *selector* to determine the message priorities to which the line applies and an *action*. The *action* field(s) are separated from the selector by one or more tabs. A maximum of 20 lines can be specified.

Selectors are semicolon separated lists of priority specifiers. Each priority has a *facility* describing the part of the system that generated the message, a dot, and a *level* indicating the severity of the message. Symbolic names can be used. An asterisk selects all facilities, while "debug" selects all levels. All messages of the specified level or higher (greater severity) are selected. More than one facility can be selected, using commas to separate them. For example:

```
*.emerg;mail,daemon.crit
```

selects all facilities at the *emerg* level and the *mail* and *daemon* facilities at the *crit* level.

Known facilities and levels recognized by *syslogd* are those listed in *syslog*(3) without the leading ''LOG_''. The additional facility ''mark'' logs messages at priority LOG_INFO every 20 minutes (this interval may be changed with the –**m** flag). The ''mark'' facility is not enabled by a facility field containing an asterisk. The level ''none'' may be used to disable a particular facility. For example:

```
*.debug;mail.none
```

sends all messages *except* mail messages to the selected file.

The second part of each line describes where the message is to be logged if this line is selected. There are five forms:

- A filename (beginning with a leading slash). The file will be opened in append mode.
- A hostname preceded by an at sign (''@''). Selected messages are forwarded to the *syslogd* on the named host.
- A comma-separated list of users. Selected messages are written to those users if they are logged in.
- An asterisk. Selected messages are written to all logged-in users.
- A |, followed immediately by a program name, which is taken to be all chars after the | up to the next tab; at least one action must follow the tab. The filter is expected to read stdin, and write the filtered response to stdout. The filter receives the source and message through stdin. A filter may also access the priority, facility and hostname via environmental variables: PRIORITY, FACILITY and FROM. The values are stored as strings defined in <sys/syslog.h>.

If the filter exits with a non-zero value, the original message is logged, as well as a message that the filter failed. The filter has a limited time (currently 8 seconds) to process the message. If the filter exits with status 0 without writing any data, no message is logged. The data to be read by the filter is not terminated with a newline, nor should the data written have a newline appended. See below for a sample filter.

Blank lines and lines beginning with '#' are ignored.

For example, the configuration file:

```
kern.debug     |/usr/sbin/klogpp          /var/adm/SYSLOG
kern.debug     |/usr/sbin/klogpp          /dev/console
user,mail,daemon,auth,syslog,lpr.debug    /var/adm/SYSLOG
kern.err       @ginger
*.emerg        *
*.alert        eric,beth
*.alert;auth.warning                      ralph
```

filters all kernel messages through /usr/sbin/klogpp and writes them to the system console and into /var/adm/SYSLOG and logs debug (or higher) level messages into the file /var/adm/SYSLOG. Kernel messages of error severity or higher are forwarded to ginger. All users will be informed of any emergency messages. The users ''eric'' and ''beth'' will be informed of any alert messages. The user ''ralph'' will be informed of any alert message or any warning message (or higher) from the authorization system.

Syslogd is started at system initialization from /etc/init.d/sysetup. Optional site-specific flags belong in /etc/config/syslogd.options. The flags are:

–f Specify an alternate configuration file.

–m Select the number of minutes between mark messages.

–d Turn on debugging. syslogd runs in the foreground and writes debugging information to stdout.

–p Use the given name for the device instead of /dev/log.

Syslogd rereads its configuration file when it receives a hangup signal, SIGHUP. To bring *syslogd* down, it should be sent a terminate signal (for example, killall –TERM syslogd).

FILTER EXAMPLE

This example shows how to use the filter mechanism. To have *ftpd*(1M) messages logged in a different file, add the following line to /etc/syslog.conf:

```
daemon,auth.debug       |/var/adm/ftpd.filt  /var/adm/ftpd.log
```

The /var/adm/ftpd.filt file is a shell script:

```
#!/bin/sh
# This filter only accepts ftpd messages
read line
set $line
case "$1" {
    ftpd\[*)
        echo "$line\c"
        exit 0
        ;;
}
exit 0
```

MESSAGE EXAMPLE

The following is an example line from the /var/adm/SYSLOG file:

```
Aug 10 10:32:53 6F:sgihost syslogd: restart
```

Each line has several parts. The date and time of the message are listed first, followed by a priority and facility code. Priorities are listed as 0-7 and facilities are listed as A-T. Reference <*sys/syslog.h*>. The source is the name of the program that generated the message. Following the source, is the message itself. Messages with "[HELP=<HELP_TAG>]" at the end, have help cards associated with them. See *sysmon(1M).*

FILES

/etc/syslog.conf	Default configuration file
/dev/log	Device read by *syslogd*
/dev/klog	The kernel log device
/usr/sbin/klogpp	Filter for kernel messages
/etc/config/syslogd.options	Command-line flags used at system startup

SEE ALSO

logger(1), syslog(3), sysmon(1M)

NAME

system – system configuration information directory

DESCRIPTION

This directory contains files (with the **.sm** suffix) which 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, the assignment of system devices such as *pipedev* and *swapdev*, as well as instructions for manually overriding the drivers selected by the self-configuring boot process. Each major subsystem may 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 upper case strings in the syntax below should be upper case 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 file name 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 **Challenge** and **Onyx** 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 manual entry. 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 IRIS 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 will attempt to read probe_size bytes (default 4) to determine if the hardware exists for the module. If the read succeeds, the hardware will be assumed to exist, and the module will be 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 may appear in any order. Blank lines may 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 defined is the historical version and will not work on newer platforms such as the **Challenge** and **Onyx** series. The second VECTOR command defined is the new version which supports the **Challenge** and **Onyx** series along with newer bus types like EISA. The second version is the preferred method since it will work across all IRIS 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 is must be a single line.) If a probe command is specified, the boot program will perform the probe sequence, as discussed above. If the sequence succeeds, the module is included. If a probe sequence is not specified, the hardware will be 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 which 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 as 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 is must be a single line.) If a probe command is specified, the boot program will perform the probe sequence, as discussed above. If the sequence succeeds, the module is included. If a probe sequence is not specified, the hardware will be assumed to exist. The bustype specifies the type of bus on which the device is connected. This would be 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 will look 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 which 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 which 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 may be referred to by expressions in part two of the **/var/sysgen/master** file.

ROOTDEV: <device>
: identifies the device containing the root file system.

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.

PIPEDEV: <device>
: identifies the device to be used for pipe space.

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 will default 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 will cause lboot to ignore the 'd' option in all master files and link all necessary modules into the kernel.

CC
LD
: are the names of the compiler and linker used to build the kernel. If absent, they default to "cc" and "ld", respectively.

CCOPTS
LDOPTS
: are option strings given to *cc*(1) and *ld*(1) respectively, to compile the master.c file and link the operating system.

FILES

/var/sysgen/system/*.sm
/usr/include/sys/edt.h

SEE ALSO

lboot(1M), **master**(4).

NAME

systune – display and set tunable parameters

SYNOPSIS

/usr/sbin/systune [–**p** rootpath] [–**n** name] [–**brfi**]

DESCRIPTION

systune is a tool that allows 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 non interactive. Interactive mode allows the user to query information about various portions of tunable parameters or to set new values for tunable parameters. To enter interactive mode, use the **–i** option. In non interactive mode, *systune* displays the values of all tunable parameters. Non interactive mode is the default.

The options are:

–**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 /.

–**n** *name*
: This option specifies an alternate kernel *name* to tune in place of **/unix**.

–**b** Both target kernel and the running system are updated with the new values that user 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.

–**r** The new values will change on the running system only. If the tunable parameter can not be changed on the running system, nothing will be affected. The default is **–b**. (See above.)

–**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 will be changed.

–**i** When *systune* is invoked in interactive mode, no parameter values are immediately displayed. Instead, you see the *systune* prompt:

```
systune->
```

The *systune* commands available in interactive mode are:

quit Quit the *systune* program 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 non interactive mode.

parameter_groupname
Display information for all the tunable parameters in this group.

parameter_name
Display information for this tunable parameter only.

parameter_name newvalue
Verify the new value and set the specified tunable parameter to the new value.

help Show all the built in commands and group names.

To display a list of the available *systune* commands and kernel parameter group names, type **`help`** at the `systune->` prompt and press **`<Return>`**. *systune* responds by listing 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, one of the parameter groups listed is *numproc. numproc* contains parameters related to the number of processes allowed to run on the system at any given time. The parameters in *numproc* are:

ncsize = 808 (0x328)
ncallout = 40 (0x28)
callout_himark = 332 (0x14c)
ndquot = 808 (0x328)
nproc = 300 (0x12c)

When you type the name of a parameter group at the *systune* prompt and press **`<Return>`**, the parameters in that group are displayed, along with their various values in decimal numerals and in hexadecimal notation. The *systune* prompt is returned at the end of the parameter display.

You can change any parameter in any group at the `systune->` prompt. Type the name of the parameter and the new value that you would like assigned to that parameter. For example, to raise the *nproc* parameter in the *numproc* parameter group from 300 to 400, type the following sequence at the `systune->` prompt and press **`<Return>`**:

```
systune-> nproc 400
```

systune responds with:

```
nproc = 300 (0x12c)
```

Do you really want to change nproc to 400 (0x190)? (y/n) **`y`**

Enter **`y`** and press **`<Return>`** to confirm your change. Next *systune* displays the following message:

```
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.

Some parameters can be changed while the system is running, and some require a new copy of the kernel to be booted. *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. (Non dynamically 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* utility. For example, some variables have preset minimum and maximum values. Any attempt to change the variable beyond these threshold values will result in an error message, and the variable will not be changed.

FILES

*/var/sysgen/mtune/**	system tunable parameters
/var/sysgen/stune	local settings for system tunable parameters

SEE ALSO

mtune(4), *stune*(4)
lboot(1M), *autoconfig*(1M)

NAME

sys_id – system identification (hostname) file

DESCRIPTION

The file **/etc/sys_id** contains the name by which the system will be known on communications networks such as the Internet and UUCP. The name can be up to 64 alphanumeric characters long and may 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 will be returned by the commands *hostname*(1) and *uname*(1), and 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

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, then **user** will be sent to the remote system as the value for the variable USER. This option may also be used with the **open** command.

–**a** Auto-login. Same as specifying –**l** with your user name. This option may 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 ′′), then there will be 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* will attempt to enable the TELNET LINEMODE option. If this fails, then *telnet* will revert 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 will relay that information. The remote system will also relay 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’’) may 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, then 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) which 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 may 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 need 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...*]
: Displays 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 will be entered.

  character
  : Disable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, then enter ‘‘character at a time‘‘ mode.

  line
  : Enable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, then 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.

?

Prints 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* will attempt to contact a TELNET server at the default port. The host specification may be either a host name (see **hosts**(4)) or an Internet address specified in the ''dot notation'' (see **inet**(3N)). The **−l** option may 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** 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) will also close a session and exit.

send *arguments*

Sends one or more special character sequences to the remote host. The following are the arguments which may be specified (more than one argument may be specified at a time):

abort

Sends the TELNET ABORT (ABORT processes) sequence.

ao

Sends 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

Sends the TELNET AYT (Are You There) sequence, to which the remote system may or may not choose to respond.

brk

Sends the TELNET BRK (Break) sequence, which may have significance to the remote system.

ec
Sends the TELNET EC (Erase Character) sequence, which should cause the remote system to erase the last character entered.

el
Sends the TELNET EL (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.

eof
Sends the TELNET EOF (End Of File) sequence.

eor
Sends the TELNET EOR (End of Record) sequence.

escape
Sends the current *telnet* escape character (initially ‘‘^]’’).

ga
Sends 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** will send the subnegotiation to request that the server send its current option status.

ip
Sends the TELNET IP (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.

nop
Sends the TELNET NOP (No OPeration) sequence.

susp
Sends the TELNET SUSP (SUSPend process) sequence.

synch
Sends 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).

?
Prints out help information for the **send** command.

set *argument value*

unset *arguments...*
The **set** command will set 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 will disable or set to FALSE any of the specified functions. The values of variables may be interrogated with the **display**

command. The variables which may be set or unset, but not toggled, are listed here. In addition, any of the variables for the **toggle** command may be explicitly set or unset using the **set** and **unset** commands.

echo
: This is 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 will cause 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, then 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
: This is 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, then 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, then 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, then 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, then 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, then 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

This is the file to which the output, caused by **netdata** or **option** tracing being TRUE, will be written. If it is set to '–', then tracing information will be written to standard output (the default).

worderase

If *telnet* is operating in LINEMODE or ''old line by line'' mode, then 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 will switch to the remote value.

?
Prints out help information for the **slc** command.

environ *arguments...*
The **environ** command is used to manipulate the variables that my 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* may be enclosed in single or double quotes so that tabs and spaces may 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 * will be sent automatically, other variables will only be sent if explicitly requested.

?
Prints out help information for the **environ** command.

?
Displays the legal **set** (**unset**) commands.

toggle *arguments...*
Toggle (between TRUE and FALSE) various flags that control how *telnet* responds to events. These flags may be set explicitly to TRUE or FALSE using the **set** and **unset** commands listed above. More than one argument may be specified. The state of these flags may be interrogated with the **display** command. Valid arguments are:

autoflush
If **autoflush** and **localchars** are both TRUE, then 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, then 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, then carriage returns will be sent as <CR><LF>. If this is FALSE, then carriage returns will be send 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 will be mapped into a carriage return followed by a line feed. 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 line feed. The initial value for this toggle is FALSE.

debug
Toggles socket level debugging (useful only to the *super*user*)*. The initial value for this toggle is FALSE.

localchars
If this is TRUE, then 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, then **quit** is sent as **abort**, and **eof**and **suspend** are sent as **eof**and **susp**, see **send** above).

netdata

Toggles the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.

options

Toggles 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 will be 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.

?

Displays 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, then 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* will print the help information for just that command.

ENVIRONMENT

Telnet uses at least the **HOME**, **SHELL**, **USER**, **DISPLAY**, and **TERM** environment variables. Other environment variables may 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

ttytype – data base of terminal types by port

SYNOPSIS

/etc/ttytype

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), login(1), terminfo(4).

NAME

versions – software versions tool

SYNOPSIS

versions [**–abndpvI**] [**–r** *root*] [**display**] [*name* ...]

versions [**–ckmpsuvxBSU**] [**–e** *pattern*] [**–i** *pattern*] [**–r** *root*]
[*listtype*] [**user**] [*name* ...]

versions [**–v**] [**–r** *root*] **remove** *name* ...

DESCRIPTION

IMPORTANT: "versions" is provided only for backwards compatibility. In reality it calls the programs *showprods*, *showfiles*, and (in the case of removing software with *versions remove*), *inst*. Users are strongly encouraged to use these programs directly, instead of versions, since the versions program will be discontinued in a future release.

To find out what command line versions would execute with a given set of arguments, use the -V option ahead of other options, e.g. **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 File System (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 eoe1 and eoe2:

```
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 will be '**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 file systems". 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 file names. The combination of single character options, *listtype*, the optional **user** argument, and optional *name*s determines the list of files that will be 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.
−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.
−i *pattern*	(include) Add a file name or file name pattern to be included in user file listings.
−e *pattern*	(exclude) Add a file name or file name pattern to be excluded from the user file listings.
−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 will cause conflicts, versions will refuse 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 may 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 will appear when you boot your system. The message will also stop appearing even if .O or .N files continue to exist after some number of reboots.

NOTES

`versions remove` will fail if there is no space in /usr to create temporary files.

FILES

/var/inst/histbinary file that contains the installation history of your machine
/var/inst/<product>binary files, one per product, available in any distribution since the last time the file systems were remade Various other files used by inst, showprods and showfiles.

SEE ALSO

showprods(1M), showfiles(1M), inst(1M), swmgr(1M),
Software Installation Administrator's Guide.

We'd Like to Hear From You

As a user of Silicon Graphics documentation, your comments are important to us. They 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 to comment on:

- 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 include the title and part number of the document you are commenting on. The part number for this document is 007-2159-004.

Thank you!

Three Ways to Reach Us

The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.

If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site: *[your_site]*!sgi!techpubs

You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964