User's Reference Manual

Introduction

User's Reference Manual

intro_urm(1)

NAME

`intro_urm -` general explanations of command descriptions

DESCRIPTION

This explanation contains general information about the individual sections of a command description. For example, notes on syntax elements and command input may be found under the section headings of the relevant command description.

The **DESCRIPTION** section of individual commands contains a description of the command's usage and effect.

SYNOPSIS

Gives an overview of the syntax of the command. The following schema contains the most important syntax elements:

`command` [{–a | –b}] [–c *optarg*] [––] [*argument*] ...

`constant width` script indicates parts which must be input exactly as they appear in the description; *italic* script denotes variables; syntax elements not entered by the user are represented by normal script.

`command`
: Command name.

–a, –b
: Options without arguments.

–c *optarg*
: Option with accompanying argument.

––
: Options separated from arguments.

argument
: Variable argument(s) for a command, e.g. a pathname, a directory, target device etc.; "–" indicates the standard input.

...
: Repeat: One or more options or arguments can be specified several times.

[□]
: Optional entry: An option or argument enclosed in square brackets may be specified. This is not mandatory.

|
: Alternative: Only one option or one of several arguments can be specified.

{□}
: Grouping: At least one option or argument from this group must be specified. Example: {*special* | *mount_point*}

OPTIONS

Displays all options and arguments that can be processed by the command.

The following rules apply for the input of options and arguments. Older commands may not comply entirely with these rules. `getopts(1)` should be used in all shell procedures to parse positional parameters and to check for legal options.

1. Option names must be one character long. (e.g. `o`, `p`, `s`).
2. All options must be preceded by "–".
3. Options with no arguments may be grouped after a single "–" (e.g. `–ps`).

4. Options with arguments must have a blank between the option and its arguments.

   For compatibility reasons the blank may be omitted from some commands. Portable shell scripts should contain a blank.
5. Option arguments must not be omitted.
6. Groups of option arguments following an option must either be separated by commas or separated by white spaces and quoted (e.g. `–o xxx,z,yy` or `–o "xxx z yy"`).
7. All command line options must precede arguments (*argument*).
8. "––" may be used to indicate the end of the option. All subsequent entries will be interpreted as arguments. The entry is necessary if its first argument begins with "–".
9. The order of options relative to one another does not matter. Options which are mutually exclusive, and repeats of options with arguments in which the arguments are evaluated in a specified sequence, are exceptions.
10. The relative order of arguments (*argument*) is significant and is determined by the command with which they appear.
11. "–" preceded and followed by white spaces should only be used to indicate standard input or standard output.

EXIT STATUS

The exit status is a numeric value which a command returns to the invoked process after its execution. It gives information about the execution of the command. If the initiated process is a shell then the exit status is stored in the shell variable `$?`.

The exit status is only described if it does not correspond to the following standards:

`0` The command was implemented correctly.

`>0` An error occurred.

DIAGNOSTICS

Important error messages are listed and explained here. Notes on avoiding errors and error recovery are also contained here. Unless otherwise specified, error messages are written to standard error output.

LOCALE

The effects of variables belonging to an internationalized environment on the command are described here.

Language, national conventions and character set may be set for the entire system or set individually using environment variables. The following variables are relevant:

`LANG`, `LC_ALL`	Entire internationalized environment
`LC_CTYPE`	Character classes and conversion from lowercase to uppercase letters
`LC_COLLATE`	Collating sequence
`LC_TIME`	Date and time specifications
`LC_MONETARY`	Currency symbols and format
`LC_NUMERIC`	Decimal point, exponentiation symbol, thousands separator
`LC_MESSAGES`	Message texts and answers to yes/no questions

To create a specific working environment you assign the name of the desired locale to the variables:

variable=localename

or

variable=language_territory.character_set

Possible values:

De_DE.646	German locale (7-bit code)
De_DE.88591	German locale (8-bit code)
En_GB.646	English locale, British English (7-bit code)
En_GB.88591	English locale, British English (8-bit code)
C, En_US.ASCII	English locale, American English (7-bit code)
En_US.88591	English locale, American English (8-bit code)
Fr_FR.88591	French locale (8-bit code)
Es_SP.88591	Spanish locale (8-bit code)
...	etc.

Further information can be found in the "Programmer's Guide: Internationalization – Localization".

EXAMPLES

These examples illustrate the main functions and the most important options of commands. Where necessary, useful complex combinations of options and arguments are described.

ENVIRONMENT VARIABLES

Environment variables that interpret the command are listed here.

NOTES

Points out borderline cases or describes known errors in the software that were not corrected. A solution is occasionally suggested.

FILES

Files which the command accesses or initializes are specified here.

SEE ALSO

This section contains reference to other commands which operate in a similar manner or interoperate with the command.

Information about syntax, options and exit status can be found in the following descriptions:

getopts(1), exit(2), wait(2), getopt(3C).

Information about command input may be found in the corresponding command interpreters:

csh(1), ksh(1), sh(1), xpg4_sh(1).

acctcom(1) - biff(1)

User's Reference Manual

acctcom(1)

NAME

acctcom - search and print process accounting files

SYNOPSIS

acctcom [*option*]... [*file*]...

DESCRIPTION

acctcom reads data in a defined format [see acct(4)] from the files specified as command-line arguments, from standard input or from a system file, and extracts the required process information. The selected information is written to standard output.

`acctcom` reports only on processes that have terminated. Information on active processes can be obtained with the `ps` command.

OPTIONS

No option specified

`acctcom` displays a list giving standard information on the individual processes. Each process record includes entries for the following fields:

```
COMMAND                    START  END    REAL     CPU      MEAN
NAME      USER    TTYNAME  TIME   TIME   (SECS)   (SECS)   SIZE(K)
```

The output is based on the *first in, first out* principle, which means that information on older processes is displayed first.

`-a` Show some average statistics about the processes selected. The statistics will be printed after the process records.

`-b` Read backwards, showing latest commands first. This option has no effect if the information is being read from standard input.

`-f` Print additional columns for the `fork`/`exec` flag and the system exit status. The numeric output for this option will be in octal.

`-h` (hog factor) Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution. This value is known as the "hog factor" and is computed as:

(total CPU time)/(elapsed time).

`-i` Print additional columns containing the I/O counts in the output.

`-k` Show total kcore-minutes, i.e. the cumulative memory usage of the process in kilobyte segments during each minute of its execution.

`-m` Show the mean core size, i.e. the mean value of the main memory requirement for each process. This option is set by default.

`-r` Show CPU factor [*user-time*/(*system-time* + *user-time*)].

`-t` Show separate system and user times.

`-v` Exclude column headings from the output.

`-l` *line*

Show only those processes that are associated with the terminal with the device entry `/dev/term/`*line*.

`-u` *user*

(user) Show only processes belonging to *user*. *user* may be specified as:

- auser ID,
- alogin name (which is converted to a user ID by the command itself),
- # (for processes executed with system administrator privileges) or
- ?(for processes associated with unknown user IDs).

`-g` *group*

(group) Show only processes belonging to *group*. The *group* may be specified in the form of a group ID or a group name.

The *time* specifications in the following options must be given in the format *hour*[:*minute*[:*second*]].

If the specified *time* is later than the present time, *time* is taken to refer to the previous day.

`-s` *time*

Select processes existing at or after *time*.

`-e` *time*

Select processes existing at or before *time*.

−S *time*
: Select processes starting at or after *time*.

−E *time*
: Select processes ending at or before *time*.

  Using the same *time* for both −S and −E shows the processes that existed at *time*.

−n *pattern*
: Show only commands with names matching the specified *pattern*. *pattern* may be in standard regular expression form [see regcmp(3G)], except that the plus sign (+) stands for one or more occurrences of the pattern.

−q
: Do not print any process records, just print the average statistics as with the −a option.

−o *file*
: Copy selected process records in the input data format to *file* without writing the information to standard output.

−H *hogfactor*
: Show only those processes with a hog factor exceeding the value specified for *hogfactor*. The method used to compute the hog factor is explained in option −h above.

−O *sec*
: Show only processes with CPU system time exceeding *sec* seconds.

−C *sec*
: Show only processes with total CPU time (system-time + user-time) exceeding *sec* seconds.

−I *chars*
: Show only processes transferring more characters than the cutoff number given by *chars*.

file
: Name of the file containing the process information to be analyzed by acctcom. This information must be in the format described under acct(4) .

  If you name more than one file, the files will be read in the order in which you name them.

  file not specified:

  If the standard input is associated with the controlling terminal but does not contain any process information, or if the standard input is connected with /dev/null as is the case with background processes of the shell, the /var/adm/pacct file is read. This file usually contains the latest process information. A busy system may need several such files, in which case they are stored under the name /var/adm/pacct*incr*, where incr is a sequential increment.

  Otherwise, standard input is read.

  The system will not generate its /var/adm/pacct*incr* files unless accounting has been started up /usr/lib/acct/startup. See the *System Administrator's Guide*.

Output

The following section lists the column titles and explains the meaning of each column in the output of acctcom. The letters following the column titles indicate the option with which each column appears in the output. "all" means that the column is displayed with all options.

Each output record represents the execution of one process.

COMMAND NAME (all)
: Name of the executed command to which the subsequent process information refers. A # is prepended to the command name if the command was executed with system administrator privileges.

USER (all)
: Login name under which the process was started.

TTYNAME (all)
: The controlling terminal of the process. If the process is not associated with a known terminal, a ? is printed in this field.

`START TIME` (all)
: Time at which the process was started.

`END TIME` (all)
: Time at which the process terminated.

`REAL (SEC)` (all)
: Total runtime for the process, in seconds.

`CPU (SEC)` (all)
: CPU system time consumed by the process, in seconds.

`CPU SYS` (`-t`)
: CPU time consumed by the system.

`(SECS) USER` (`-t`)
: CPU time consumed by the user.

`MEAN SIZE(K)` (all)
: Mean memory size for the process.

`F` (`-f`)
: The `fork`/`exec` flag: 1 for `fork` without `exec`.

`STAT` (`-f`)
: System exit status.

`HOG FACTOR` (`-h`, `-H`)
: Hog factor.

`KCORE MIN` (`-k`)
: Cumulative memory usage of the process in kilobyte segments during each minute of its execution.

`CPU FACTOR` (`-r`)
: CPU factor.

`CHARS TRNSFD` (`-I`)
: Number of characters transferred.

`BLOCKS READ`
: Total number of blocks read and written.

EXAMPLES

Example 1

You need to look at standard accounting information on all the latest terminated processes:

```
$ acctcom
ACCOUNTING RECORDS FROM:  Wed Feb  6 10:22:30 1991
COMMAND                 START    END       REAL    CPU   MEAN
NAME     USER  TTYNAME  TIME     TIME      (SECS) (SECS) SIZE(K)
#accton  root  tty01    10:22:30 10:22:30   0.13   0.03   8.17
turnacct root  tty01    10:22:29 10:22:30   1.27   0.12   0.92
rm       root  tty01    10:22:30 10:22:30   0.03   0.03   1.50
rm       root  tty01    10:22:31 10:22:31   0.02   0.02   2.25
startup  root  tty01    10:22:28 10:22:30   2.51   0.20   0.53
acctcom  root  tty01    10:23:21 10:23:21   0.39   0.20   1.43
getopt   root  tty01    10:24:09 10:24:09   0.05   0.03   8.00
expr     root  tty01    10:24:09 10:24:09   0.07   0.03   0.67
...
```

If the output needs to be paged, you can pipe the output of `acctcom` to the `pg` command:

```
$ acctcom | pg □
...
```

Example 2

You now want to select the accounting information for all processes with CPU system time exceeding one

second, save this information in a file named `boredom`, and then print the contents of that file.

```
$ acctcom –O 1 –o boredom
ACCOUNTING RECORDS FROM:  Wed Feb  6 10:22:30 1991
$ acctcom boredom
ACCOUNTING RECORDS FROM:  Wed Feb  6 10:24:10 1991
COMMAND                    START     END        REAL   CPU   MEAN
NAME      USER   TTYNAME   TIME      TIME      (SECS) (SECS) SIZE(K)
find      root    pts/0    10:24:10  10:25:48   98.40   1.66  0.17
acctdusg  root    pts/0    10:24:10  10:26:21  131.20   1.36  0.24
```

FILES

`/etc/passwd`
: File containing login names/user IDs.

`/etc/group`
: File containing group names/IDs.

`/var/adm/pacct`*incr*
: File containing the current process information. Busy systems may need several such files, in which case they are numbered sequentially and can be distinguished by the suffix *incr*.

SEE ALSO

ps(1), su(1), acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), fwtmp(1M), runacct(1M), acct(2), regcmp(3G), acct(4), utmp(4).

addbib(1)
BSD Compatibility Package

NAME

addbib - create or extend a bibliographic database

SYNOPSIS

/usr/ucb/addbib [–a] [–p *promptfile*] *database*

DESCRIPTION

When addbib starts up, answering y to the initial Instructions? prompt yields directions; typing n or RETURN skips them. addbib then prompts for various bibliographic fields, reads responses from the terminal, and sends output records to *database*. A null response (RETURN) means to leave out that field. A "–" (minus sign) means to go back to the previous field. A trailing backslash allows a field to be continued on the next line. The repeating Continue? prompt allows the user either to resume by typing y or RETURN, to quit the current session by typing n or q, or to edit *database* with any system editor (vi, ex, ed).

OPTIONS

–a Suppress prompting for an abstract; asking for an abstract is the default. Abstracts are ended with a <CTRL-D>.

–p *promptfile*
Use a new prompting skeleton, defined in *promptfile*. This file should contain prompt strings, a TAB, and the key letters to be written to the *database*.

USAGE

Bibliography Key Letters

The most common key letters and their meanings are given below. addbib insulates you from these key letters, since it gives you prompts in English, but if you edit the bibliography file later on, you will need to know this information.

%A Author's name
%B Book containing article referenced
%C City (place of publication)
%D Date of publication
%E Editor of book containing article referenced
%F Footnote number or label [supplied by refer(1)]
%G Government order number
%H Header commentary, printed before reference
%I Issuer (publisher)
%J Journal containing article
%K Keywords to use in locating reference
%L Label field used by –k option of refer(1)
%M Bell Labs Memorandum (undefined)
%N Number within volume
%O Other commentary, printed at end of reference
%P Page number(s)
%Q Corporate or Foreign Author (unreversed)

`%R` Report, paper, or thesis (unpublished)

`%S` Series title

`%T` Title of article or book

`%V` Volume number

`%X` Abstract – used by `roffbib(1)`, not by `refer(1)`

`%Y`, `%Z`
Ignored by `refer(1)`

SEE ALSO

`ed(1)`, `ex(1)`, `indxbib(1)`, `lookbib(1)`, `refer(1)`, `roffbib(1)`, `sortbib(1)`, `vi(1)`.

User's Reference Manual

apropos(1)

NAME

`apropos` - locate commands by keyword lookup

SYNOPSIS

`apropos` *keyword ...*

DESCRIPTION

`apropos` shows which manual sections contain instances of any of the given keywords in their title. Each word is considered separately and the case of letters is ignored. Words which are part of other words are considered; thus, when looking for "compile", `apropos` will find all instances of "compiler" also.

Try

```
apropos password
```

and

```
apropos editor
```

If the line starts "*filename* (*section*) ..." you can do "`man` *section filename*" to get the documentation for it. Try

```
apropos format
```

and then

```
man 3s printf
```

to get the manual page on the subroutine `printf`.

`apropos` is actually just the `–k` option to the `man(1)` command.

FILES

`/usr/share/man/whatis`
data base

SEE ALSO

`man(1)`, `whatis(1)`, `catman(1M)`.

User's Reference Manual

ar(1)

NAME

`ar` - archive and library maintainer

SYNOPSIS

`ar` [`-V`] *mainkey* [*modifier*]... [*posname*] [`--`] *afile* [*filename*]...

DESCRIPTION

`ar` is used to maintain archives (libraries).

`-V` `ar` prints its version number on standard error.

In specific terms, you can use `ar` to perform the following file management functions:

Function	Main keys
create an archive file	`-q` or `-r`
quickly append a file to the end of the archive	`-q`
place/replace a file in the archive	`-r`
delete a file from the archive	`-d`
move a file within the archive	`-m`
print the contents of a file	`-p`
list the files in the archive	`-t`
copy (extract) a file from the archive	`-x`

Structure of an archive

An archive is a single file that combines several files into one. By convention, the name of an archive file ends with the suffix `.a`. The purpose of an archive file is to enable the collective maintenance of a group of related files. Archives make it easier to maintain files, since you will often only need to specify the archive file instead of individually listing all its elements.

The constituents of an archive are typically object modules that usually form part of the same program or suite of programs.

The magic string and the file headers used by `ar` consist of printable ASCII characters. If an archive is entirely composed of printable files, the entire archive is printable.

When `ar` creates an archive, it creates headers in a format that is portable across all machines (for the portable archive format and structure [see `ar`(4)].

If the archive contains at least one object file, `ar` creates and maintains an archive symbol table. This symbol table is used by the link editor `ld` to effect multiple passes over the archive in an efficient manner. The archive symbol table is stored in a specially named file that is always the first file in the archive. This file is never listed or printed out like other archive files and is not accessible to the user. Whenever the `ar` command is used to create or update the contents of an archive, the symbol table is rebuilt. The symbol table can also be rebuilt by calling `ar` with the `-s` option.

OPTIONS

Main keys

When you call `ar`, you must specify precisely one main key (`-d`, `-m`, `-p`, `-q`, `-r`, `-t` or `-x`), optionally followed by one or more arguments.

`-d` `ar` deletes the specified files from the archive. If no files are specified, no action is taken.

`-m` `ar` moves the specified files within the archive.

With *posname*:

The files are placed after (`-a`) or before (`-b` or `-i`) the file named *posname* (see *posname*).

Without *posname*:

The files are appended to the end of the archive.

`-p` `ar` prints the contents of the specified files. If no files are given, the contents of all files are printed.

`-q` The specified files are added to the archive "quickly", i.e.:

- if the archive already exists, `ar` appends the named files to it without checking whether they are already present in the archive;
- if the archive does not yet exist, it is created.

No *posname* arguments are permitted.

The `-q` option is useful when creating a large archive piece-by-piece.

`-r` (replace) This option has three different effects, depending on whether the specified archive exists and whether it contains the named files:

- If the archive exists and contains the files, `ar` replaces the named files.
- If the archive exists but is missing one or more of the named files, `ar` adds the missing files to the archive.
- If the archive does not exist, `ar` creates the archive from the named files.

With modifier `-u`:

`ar` replaces a file in the archive only if the named file has a later date of last modification than the version already in the archive.

With *posname*:

Files not yet in the archive are placed after (`-a`) or before (`-b` or `-i`) the file *posname* (see *posname*).

Without *posname*:

Files not yet in the archive are appended to the end of the archive.

`-t` `ar` prints a table of contents of the archive file. If no files are specified, all files in the archive are listed; otherwise, only the named files.

`-x` `ar` copies (extracts) files from the archive.

If no files are specified, all files in the archive are extracted. The archive itself remains unaltered.

Key modifiers

`-c` (create) The message that `ar` usually issues when creating an archive is suppressed.

`-C` (C – create) Protection against overwriting. `-C` prevents files extracted using `-x` from replacing like-named files.

`-l` (l – local) Temporary files are placed in the current directory.

`-s` (symbol table) `ar` regenerates the archive symbol table (even if the archive contents have not been modified). This modifier is useful to restore the archive symbol table after the `strip` command has been used on the archive [see `strip(1)`].

`-T` (T – truncate) `ar` truncates filenames if they are too long for the current file system. For security reasons, this option should only be used together with `-C`.

`-T` not specified:

Files whose names are too long are not extracted by `ar`.

`-u` (update) See main key `-r`.

–v (verbose) ar reports each action it takes (adding files to the archive, moving them within the archive, extracting them from the archive, etc.). If –v is used with the –t key, ar displays a long listing similar to that produced by the ls –l command [see ls(1)].

posname
The *posname* argument specifies the position at which a file is inserted into the archive. *posname* may be:

–a *posfile*
ar places the file after the named *posfile*.

–b *posfile*
–i *posfile*
ar inserts the file before the named *posfile*.

posfile is the name of a file already in the archive.

–– If *afile* begins with a dash (–), the end of the command-line options must be marked with ––.

afile Name of the archive to be created or processed.

filename
Name of the file to be listed, printed out, added to, extracted from, or moved within the archive.

More than one file may be named on the command line. If you specify the same file more than once when calling ar with the –q or –r key, ar will enter the named file into the archive as often as specified. If a number of identically named files are contained in the archive, and you use ar without specifying a position, the first file with the given name will be accessed (rather than the most recent or oldest file, for example).

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

The LC_TIME environment variable governs the format of date and time strings in archive content listings produced in conjunction with the v modifier.

If LC_MESSAGES or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables were set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

Quickly creating an archive file:

```
$ ar –q –v afile.a atoi.o itoa.o
ar: creating afile.a
a – atoi.o
a – itoa.o
```

ar creates the archive afile.a from the files atoi.o and itoa.o. The v modifier tells ar to display the names of the files as they are added.

Example 2

Placing a new file at a specified position in an archive:

```
$ ar –r –v –b atoi.o afile.a atof.o
a – atof.o
```

ar copies atof.o into the archive file afile.a, placing it before atoi.o.

Example 3

Printing the table of contents of an archive:

```
$ ar –t –v afile.a
rw-r--r-- 104/     1      2276 Jul 13 12:17 1990 atof.o
rw-r--r-- 104/     1       759 Jul 13 12:17 1990 atoi.o
rw-r--r-- 104/     1      1280 Jul 13 12:17 1990 itoa.o
```

Example 4

Extracting (copying) a file from an archive:

```
$ ar –x –v afile.a atoi.o
x – atoi.o
```

The file `atoi.o` is copied from the archive `afile.a` into the working directory. The copy is also called `atoi.o`.

SEE ALSO

`cpio`(1), `ld`(1), `lorder`(1), `pax`(1), `strip`(1), `tar`(1), `a.out`(4), `ar`(4).

User's Reference Manual

arch(1)
BSD Compatibility Package

NAME

`arch -` display the architecture of the current host

SYNOPSIS

```
/usr/ucb/arch
```

DESCRIPTION

The `arch` command displays the architecture of the current host system.

SEE ALSO

`mach(1)`, `uname(1)`.

User's Reference Manual

at(1)

NAME

`at -` execute commands at a later time

SYNOPSIS

Format 1

`at` [`-m`] [`-f` *script*] [`-q` *queue*] `-t` *time*

Format 2

`at` [`-m`] [`-f` *script*] [`-q` *queue*] [`--`] *time*

Format 3

`at -l` [*jobnumber*] ...

Format 4

`at -r` *jobnumber* ...

DESCRIPTION

The `at` command

- reads commands from standard input or a shell script and executes them at a later time specified by the user (**Format 1** and **2**)
- lists on standard output any jobs scheduled with `at` or `batch` [see `batch(1)`] which have not yet been processed (**Format 3**)
- removes jobs previously scheduled with `at` or `batch` (**Format 4**).

The output from **Format 1** and **Format 2** is sent to the user by `mail` unless the standard output and standard error output of the commands to be executed have been redirected. The environment variables, the current directory, the permissions for new files [see `umask(1)`] and the maximum permissible file size [see `ulimit(1)`] are retained, but open files and priorities are lost, and the `trap(1)` command (shell built-in for catching signals) is deactivated.

`at` writes the job number and the schedule time to standard error. Jobs scheduled with `at` are run even if the user who scheduled them logs out of the system.

Before the call

If the file `/etc/cron.d/at.allow` exists, you can only use `at` if your login name appears in it.

If the file `/etc/cron.d/at.allow` does not exist, you can only use `at` if your login name does `not` appear in the file `/etc/cron.d/at.deny`.

If neither `/etc/cron.d/at.allow` nor `/etc/cron.d/at.deny` exists, only the system administrator is allowed to use `at`.

If only an empty `deny` file exists, for example, everyone is allowed to use `at`.

Only the system administrator is allowed to create and modify the `allow` and `deny` files. Each line in these files contains precisely one login name.

OPTIONS

Format 1: Execute commands at a later time

`at` [`-m`] [`-f` *script*] [`-q` *queue*] `-t` *time*

`-f` *script*

`at` reads the commands to be executed from the specified shell script.

`-f` not specified:

`at` reads the commands to be executed from the standard input.

You can specify a number of commands, separated from one another by a semicolon (;) or a newline character. A command list created in this way runs under a job number. <CTRL-D> exits the command list.

`-m` Sends mail to the user after a job has been completed, indicating that the job is finished. Mail is sent only if the job has not already generated a mail message.

`-q` *queue*

The `-q` option is used to assign a job to a specific queue in the `/var/spool/cron` directory.

The values accepted for *queue* are:

`a` The default queue for jobs scheduled with `at`.

`b` The default queue for jobs scheduled with `batch`.

`d - z`

User-defined queues [cf. `queuedefs(4)`].

`-t` *time*

Specifies the execution time for the commands. *time* is specified as follows:

[[*CC*]*YY*]*MMDDhhmm*[. *SS*]

CC The first two digits of a date (century): `19` or `20`

CC not specified:

If the two-digit year notation *YY* is in the range `70` to `99`, the 20th century is assumed; if it is in the range `00` to `37`, the 21st century is assumed. All other entries are rejected.

YY Two-digit year specification

Only the digits >`69` and <`38` may be entered. If the year number specified is <`38`, this is beyond the year 2000.

YY not specified: The current year is assumed.

MM Two-digit month specification (01 through 12)

DD Two-digit day specification (01 through 31)

hh Two-digit hour specification (00 through 23)

mm Two-digit minute specification (00 through 59)

SS Two-digit second specification (00 through 61)

The values `60` and `61` are intended for leap seconds.

SS not specified: The value 0 seconds is assumed.

Format 2: Execute commands at a later time

`at` [`-m`] [`-f` *script* [`-q` *queue*] [`--`] *time*

The options are described under **Format 1**.

`--` If *time* begins with a dash (-), the end of the command-line options must be marked with `--`.

time Specifies the execution time for the commands. *time* is specified as follows: *Time* [*Date*] [+*Increment*]

Time *digits*[*suffix*] or *special_name*

digits

[*h*]*h* 1 and 2-digit numbers are interpreted as hours.

hhmm 4-digit numbers are interpreted as hours and minutes.

[*h*]*h* : [*m*]*m*

Digits separated by a colon are interpreted as hours and minutes.

suffix

`am` Interpreted as before 12 noon

`pm` Interpreted as after 12 noon

`am`, `pm` omitted
: Interpreted as 24-hour clock

`utc` Interpreted as Greenwich Mean Time

`zulu` Interpreted as Greenwich Mean Time

special_name

`noon`
`midnight`
`now`
: `at now` on its own will result in the error message "Too late".

Date *month day*[,*year*] or *weekday* [`nextweek`] or *special_day*

month

`jan`
`feb`
`mar`
`apr`
`may`
`jun`
`jul`
`aug`
`sep`
`oct`
`nov`
`dec`

day A number between 1 and 31, depending on how long the month is

year Number defining the year to which the date applies

`nextyear`
: selects the following year

year not specified: The current year is assumed.

weekday

`mon`[`day`]
`tue`[`sday`]
`wed`[`nesday`]
`thu`[`rsday`]
`fri`[`day`]
`sat`[`urday`]
`sun`[`day`]

`nextweek`
: selects the following week

special_day

`today`
`tomorrow`
`nextday`

`nextday` means that the job is executed a full day later.

If the specified time lies before the current time, `at` interprets the following day as the current day. For example, if the job `at 10 nextday` were scheduled at 11:00am on 7/1/91, the job would be executed at 10:00am on 7/3/91; however, if `at 14 nextday`

were specified at the same time, execution would begin at 14:00 hours on 7/2/91.

Date not specified:

- corresponds to `today` if the specified time (rounded to the nearest minute) lies after the current time.
- corresponds to `tomorrow` if the specified time (rounded to the nearest minute) lies before the current time.
- corresponds to `now` if the specified time (rounded to the nearest minute) is the same as the current time.

+Increment

Is a positive integer that must be followed by one of the following units of time:

`minute`[`s`]

`hour`[`s`]

`day`[`s`]

`week`[`s`]

`month`[`s`]

`year`[`s`]

Examples: `at` can typically be specified in the following ways:

```
at 0815am jan 24 □
at 8:15am jan 24 □
at 5pm friday □
at now +1hour
```

Format 3: List jobs yet to be processed

`at` `−l` [*jobnumber*] ...

`−l` [*jobnumber*]

`at` lists the specified *jobnumber* if the corresponding job has not yet been processed. *jobnumber* is the number that is reported on standard error when a job is scheduled with `at`, `batch` or `cron`.

jobnumber not specified:

`at` lists all jobs that are yet to be processed, together with their job numbers.

Format 4: Remove jobs

`at` `−r` *jobnumber* ...

`−r` *jobnumber*

`at` removes a job previously scheduled with `at` or `batch`, where *jobnumber* is the number reported on standard error when a job is scheduled with `at` or `batch`. You can specify more than one job number, using blanks to separate them. Only the system administrator is allowed to remove another user's jobs.

ERROR MESSAGES

The commonest error messages are:

`at: bad date specification`

You have entered the date in an incorrect format.

`at: too late`

You have specified "now" as the value for *Time*. Since "now" refers to the time at which `at` is called, it is always too late as the time for command execution.

`at: you are not authorized to use at. Sorry.`

See **Before the call**.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

The LC_TIME environment variable governs the format of date and time strings.

If LC_MESSAGES or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables were set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

The current date and the string April Fool! are to be displayed on terminal tty013 at 9 o'clock in the morning on the 1st of April.

```
$ at 9am apr 1 <RETURN>
echo `date`": April Fool!" >> /dev/tty013 <RETURN>
<CTRL-D>
```

FILES

/etc/cron.d/at.allow
: List of login names with permission to use at. One login name is entered per line.

/etc/cron.d/at.deny
: List of login names explicitly denied permission to use at. One login name is entered per line.

/var/spool/cron/atjobs
: Directory containing a separate file for each at job which has not yet been executed. Each at job is allocated a file of its own with the file name *jobnumber*.a.

/etc/cron.d/queuedefs
: File containing scheduling information.

SEE ALSO

atq(1), atrm(1), batch(1), calendar(1), date(1), kill(1), mail(1), nice(1), ps(1), sh(1), sort(1), ulimit(1), cron(1M), getdate(3C), queuedefs(4), environ(5).

User's Reference Manual

atq(1)

NAME

`atq` - list jobs scheduled with at or batch (at queue)

SYNOPSIS

`atq` [*option*] [*login-name*]...

DESCRIPTION

`atq` lists jobs that were scheduled with `at` or `batch` but have not yet been executed.

If you are not a privileged user, you can only list your own jobs with `atq`; system administrators can list all jobs in the queue.

OPTIONS

No option specified

The jobs are listed in chronological order of execution.

`-c` The jobs are listed in the order they were created.

`-n` `atq` displays only the total number of jobs currently in the queue.

Only for the system administrator

login_name

Only those jobs belonging to the named user are displayed. You may specify more than one login name.

login_name not specified:

If you invoke `atq` as the system administrator without specifying a *login_name*, all jobs in the queue are displayed.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_TIME` environment variable governs the format of date and time strings.

If `LC_XXXX` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

```
$ atq
Rank     Execution Date   Owner    Job         Queue  Job Name
1st   Jun 6, 1990 18:00  anna   644709600.a    a      stdin
2nd   Dec 7, 1990 20:30  anna   644711400.a    a      stdin
```

FILES

`/var/spool/cron`

Directory containing the jobs that are still to be processed.

SEE ALSO

`at(1)`, `atrm(1)`, `batch(1)`, `crontab(1)`, `cron(1M)`.

User's Reference Manual

atrm(1)

NAME

`atrm -` remove jobs scheduled with at or batch (at remove)

SYNOPSIS

`atrm` [*option*]... *argument*...

DESCRIPTION

`atrm` removes jobs that were scheduled with `at` or `batch` but have not yet been executed.

If you are not a privileged user, you can only remove your own jobs with `atrm`; system administrators can remove any jobs from the queue.

OPTIONS

`–a` (all) `atrm` removes all your jobs from the queue.

If you invoke this command as system administrator, the entire queue will be flushed.

This option must be specified without an *argument*.

`–f` (force) All messages regarding the removal of the specified jobs are suppressed.

`–i` (interactive) `atrm` expects you to individually confirm each job to be removed. Only the jobs you confirm will be removed.

argument

argument can be a login name or a job number. `atrm` removes each job number you specify and/or all jobs belonging to the specified users.

As an ordinary user you can only remove jobs that belong to you; only the system administrator can remove jobs belonging to other users.

A list of scheduled jobs and associated job numbers can be displayed by using `atq`.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_TIME` environment variable governs the format of date and time strings.

If `LC_XXX` or `LC_YYY` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables were set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

After using `atq` to list all unexecuted jobs that were scheduled with `at` or `batch`, you remove the job with job number 644711400.a.

```
$ atq
Rank   Execution Date    Owner    Job        Queue  Job Name
1st   Jun 6, 1990 18:00  anna   644709600.a   a      stdin
2nd   Dec 7, 1990 20:30  anna   644711400.a   a      stdin
$ atrm 644711400.a
644711400.a removed
```

FILES

`/var/spool/cron`

Directory containing the jobs that are still to be processed.

SEE ALSO

at(1), atq(1), batch(1), crontab(1), cron(1M).

User's Reference Manual

awk(1)

NAME

`awk, nawk -` pattern-directed scanning and processing language

SYNOPSIS

`awk` [–F*c*] [–v *initialization*] ... [––] *prog* [*initialization ...*] [*file ...*]

DESCRIPTION

`awk` is a programmable text manipulation system. When you call `awk` you specify an `awk` program it is to execute and the files it is to process. The actions defined in the program are then performed on the basis of the specified files. `awk` does not alter its input files. The results of the actions it performs are by default written on standard output.

`awk` offers the following advantages over text manipulation programs such as `egrep` and `sed`:

- `awk` operates on one record at a time. As with `egrep` and `sed`, an input record is defined as one line by default; but with `awk` you can change this setting and define some other unit of text as the record.
- Each input record is split into fields which can be accessed individually.
- A pattern (selection criterion) may be a condition defined by the logical combination of extended regular expressions and relational operators.
- You can program any actions that you require. `awk` is a high-level C-like programming language.

A detailed description of `awk` is provided below in the following sections:

- Typical `awk` applications
- Structure of an `awk` program
- Operation of the `awk` command
- The input file (records, fields, special variables)
- Basic elements of the `awk` language (comments, constants, variables)
- Expressions
- Patterns
- Actions (control flow statements, functions).

OPTIONS

–F*c* Defines the field separator character for the input record (input field separator).

c Regular expression that defines a character to be interpreted as the input field separator. Separators do not form part of the fields.

Note:

To be able to use "t" as the input field separator, you must specify it as follows on the `awk` command line or in the `BEGIN` section of the `awk` program:

`awk –F[t] ...` or `BEGIN {FS=t ...}`

–F*c* not specified:

Blanks and tabs act as field separators.

–v *initialization*

Assignments in the form: *var*=*value*.

The *var* variable which appears in the program is initialized to *value*.

var Name of the variable to be initialized.

value

Initial value to be assigned to *var*. *value* can be defined in exactly the same way as an

environment variable on shell level.

`--` If *prog* begins with a dash (-), the end of the command-line options must be marked with `--`.

prog `awk` program argument. Possible forms for *prog* are:

'*awk-program*', i.e. an `awk` program written on the command line, or:

`-f` *progfile*, i.e. the name of a file containing an `awk` program.

'*awk_program*'

An `awk` program written on the command line.

You should always enclose the `awk` program in single quotes in order to prevent the shell from interpreting metacharacters. If the program is more than one line long, you must escape the newline character with a backslash.

Example:

Process the file named `input` and display all lines that have a "0" in the third field:

```
$ awk '$3 == 0' input
```

`-f` *progfile*

The `awk` program is located in the file named *progfile*.

You can specify a number of `awk` programs. Each `awk` program specified must be preceded by `-f`. Where multiple specifications are made, `awk` processes the files in the specified order.

initialization

Assignments in the form: *var*=*value*.

The *var* variable (whether it appears in the `awk` program or not) is initialized to *value*. *initialization* and *file* may be specified in any order. The assignment is made at the time when the named file is opened.

Note:

The `$` variables (see **BASIC ELEMENTS OF THE AWK LANGUAGE** below) cannot be initialized in this way.

var Name of the variable to be initialized. The name must not begin with `$`.

value

Initial value to be assigned to *var*. *value* can be defined in exactly the same way as an environment variable on shell level.

file Name of the text file to be processed. You may list more than one file if you wish. Files are read in the order in which they are listed. If *file* is a dash (-), `awk` reads from standard input.

file not specified:

`awk` reads from standard input. `awk` reads input one record at a time, processes it, and after each line outputs the result for that record. Hitting <CTRL-D> terminates your input.

Known bugs

The assignment results via *initialization* – unlike the description under *initialization* (see above) – even **before** you call `awk`. With SINIX V5.43 or later, an assignment is still only possible before you begin `awk` via "`-v` *initialization*", as defined in XPG4.

TYPICAL AWK APPLICATIONS

`awk` is a tool which makes text manipulation tasks easy to accomplish. Typical applications for `awk` include:

- selectively extracting data from files
- checking the contents of files
- performing calculations on the data in a file
- changing the format of input data.

Using four simple examples, this section demonstrates how `awk` can be used.

A file called `supplies` contains a list of office supplies. It includes the name of each article, along with its quantity and unit price:

```
Pencil      100     0.60
Table         5   345.00
Lamp         20    79.80
Paper        75     1.00
Diskette   1000     2.40
Envelope   1500     0.20
```

Example 1

Select all articles with a quantity greater than 100:

```
$ awk '$2 > 100 {print}' supplies
Diskette   1000     2.40
Envelope   1500     0.20
```

With `$2` you access the second field of a line, which in this case is the quantity of each article. If the quantity is greater than 100, the condition is fulfilled, and the `print` function is executed. Since no arguments were specified for `print`, the whole line is output.

Example 2

Calculate the total price for all articles with a quantity greater than 100 and print this total along with the article name:

```
$ awk '$2 > 100 {print $1 "\t" $2*$3}' supplies
Diskette        2400
Envelope        300
```

Three arguments are entered for the `print` function in this example. The following is output:

`$1` article name (first field)

`\t` tab character

`$2*$3`
quantity (second field) times unit price (third field)

Example 3

Include a heading in the output:

```
$ awk 'BEGIN    {print "Article \tTotal"}
>      $2 > 100 {print $1 "\t\t" $2*$3}' supplies
Article         Total
Diskette        2400
Envelope        300
```

This example illustrates the use of the `BEGIN` pattern. `awk` executes the action after `BEGIN` only once, i.e. when the program is started. The heading is therefore printed only once at the beginning.

Example 4

Print a grand total of all amounts at the end.

For this purpose we use a variable called `sum`, which is initialized to zero in the `BEGIN` pattern. The product of column 2 and column 3 is calculated for each line, and all the products are summed up:

```
$ awk 'BEGIN    {sum=0; print "Article \tTotal"}
>     $2 > 100 {print $1 "\t\t" $2*$3; sum += $2*$3}
>     END      {print "\nGrand total: " sum} ' supplies
Article         Total
Diskette        2400
Envelope        300
Grand total:    2700
```

This example demonstrates the use of the `END` pattern. `awk` executes the action after `END` only once, i.e. before termination of the program. The grand total of all subtotals is therefore printed just once at the end.

STRUCTURE OF AN AWK PROGRAM

An awk program can consist of a BEGIN section, a main section, and an END section, structured as shown below:

[BEGIN {*action*}]

[[*pattern*] {*action*}

[END {*action*}]

pattern

The *pattern* indicates which data is to be selected from the input files (see **Patterns** below).

action

The *action* indicates what to do with data that matches the pattern (see **Actions** below).

function-definition

A *function_definition* enables you to define your own functions (see **FUNCTIONS** below).

At least one of the three sections (*pattern*, *action* or *function-definition*) must be present.

In a *pattern* {*action*} pair, either the pattern or the action can be omitted. If the action is omitted, each line that matches the pattern is output; omitting the pattern causes the action to be performed on all lines.

The definition of a user-defined function may appear at any position in the main section.

Each of the following elements must be located at the start of a line (following any number of blanks or tabs):

- the BEGIN section,
- the [*pattern*] {*action*} and *pattern* [{*action*}] pairs,
- the function definitions,
- the END section.

OPERATION OF THE AWK COMMAND

awk executes the awk program that is specified by the user, proceeding in the following sequence:

1. Initial processing

   The first step performed by awk is to initialize any variables that may have been defined. If there is a BEGIN section including an *action*, awk then executes the action specified there. The action in the BEGIN section is executed just once, before the first line is processed.

2. File processing

   Next awk processes the specified input files by reading the input records sequentially. For each input record, awk tries to match each pattern in the order that is specified in the awk program. If a pattern is matched, i.e. the selection criterion is fulfilled, the associated action is performed.

   If no pattern is specified for an action, awk performs the action for every record.

   If no action is specified for a pattern, the default action is to output (print) the record.

   Multiple input files are processed in the specified order.

3. Final processing

   When all the specified files have been processed, awk performs the action in the END section, if one has been included. awk then exits.

THE INPUT FILE

An input file consists of records that are subdivided into fields.

Records

Records are separated by a record separator. The record separator does not form part of a record. By default, a record is one line, and the record separator is the newline character. However, you do have the option of changing this setup by assigning any single character to the special variable RS (Record Separator). If you specify a string of characters as a value for RS, only the first character will be taken into account. The ordinal number of the current record is available in the variable NR (Number of Record). If there is more than one input file, NR counts from the start of the first file to the end of the last one. The special variable $0 addresses

the whole of the current record. Further information on variables is provided in the section on **BASIC ELEMENTS OF THE AWK LANGUAGE**.

Fields

Each record is split into fields separated by one or more field separators. The default field separator is white space (any sequence of tabs and blanks), but you do have the option of changing this by assigning any other character to the special variable `FS` (Field Separator). You can make this assignment either in the `awk` program or by using option `-F` on the command line. The value assigned to `FS` is interpreted as an extended regular expression [see `expressions(5)`].

Example 1

To define the characters *x* and *y* as alternate field separators:

syntax on the `awk` command line: `-F"[xy]"`

syntax in the `awk` program: `FS="[xy]"`

Example 2

To define the field separator as one or more occurrences of the character *x*:

syntax on the `awk` command line: `-Fx+`

syntax in the `awk` program: `FS=x+`

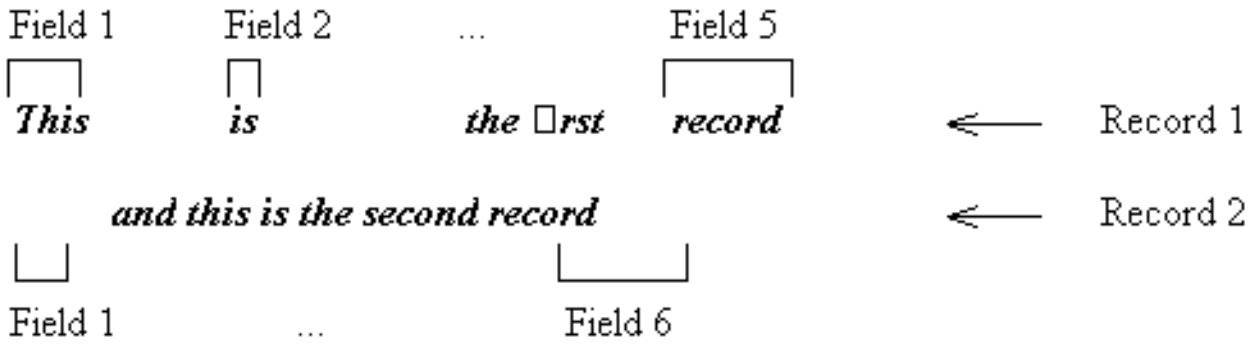
The default setting (any sequence of blanks and tabs) can be expressed by the regular expression `"[ \t]+"`, where stands for a blank, and `\t` represents a tab.

Note that the newline character is always interpreted as a field separator, regardless of the value assigned to `FS`!

The number of fields in the current record is stored in the variable `NF` (Number of Fields). Individual fields of the current record are addressed by the predefined variables `$1`, `$2`, to `$NF`. Further information on variables is provided in the section on **BASIC ELEMENTS OF THE AWK LANGUAGE**.

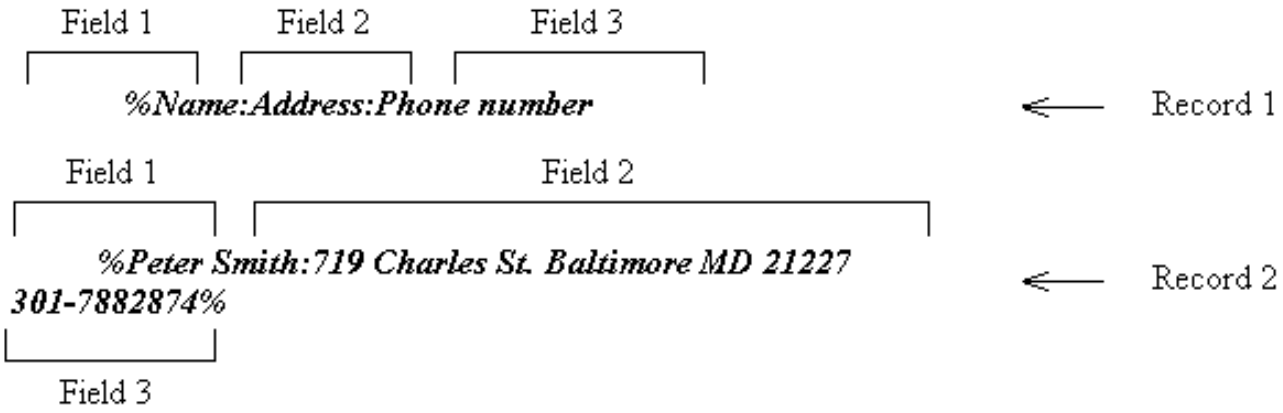
Example 1

Default setup



Example 2

Modified setup: `RS="%"; FS=":";`



Rules for record and field separators

- Default settings for record separators
  - The default record separator is the newline character.
  - If the null string is assigned to `RS` (`RS=""`), the file is treated as a single record. If several files are specified, each file will consist of a single record (which means that the ultimate value of `NR` will be equal to the number of files).
- Default settings for field separators
  - If the record separator is newline, the field separator defaults to blanks and tabs.
  - If the record separator is not a newline, the newline character **always** counts as a field separator, regardless of which character has been explicitly defined as the field separator (see **Fields** above, *Example 2*).
  - If you explicitly assign a blank to `FS`, either with `–F"␣"` on the `awk` command line or by using the assignment `FS="␣"`, then blanks **and** tab characters are treated as field separators.
  - On the other hand, if you explicitly assign the tab character to `FS` (`FS="\t"`), then only the tab character is treated as the field separator and not the blank.
- Leading field separators and field separator strings
  - The following applies to blanks, tabs and newlines as field separators:
    1. Leading field separators are ignored.
    2. Multiple occurrences of a field separator are treated as a single field separator (see **EXAMPLES**, *Example 9*).
  - For all other field separators, leading field separators are counted. In multiple occurrences of a field separator, each character is counted separately. Thus two consecutive field separators are deemed to have an empty field between them (see **EXAMPLES**, *Example 10*).
- Changing separators

  If you need a number of different record separators in one file, you can change `RS` within the `awk` program. The new record separator comes into effect as soon as the assignment to `RS` has been implemented. Similarly, you can change `FS` within the `awk` program, should you require a number of different field separators in one file. The new field separator comes into effect as soon as the assignment to `FS` has been implemented.

Special variables for the input file

The following table contains all special `awk` variables pertaining to the input file. The value `awk` usually assigns to these variables is indicated in the second column.

Variable	Value set by `awk`

FILENAME	Name of the current input file, – for standard input
FS	Input field separator (default: any sequence of blanks and tabs)
NF	Number of fields in the current record
NR	Ordinal number of the current record from start of input
FNR	Ordinal number of the current record in the current file
RS	Input record separator (default: newline)
$0	Current record
$1	First field of the current record
$2	Second field of the current record
...	
$NF	Last field of the current record

You can change these variables within an awk program if you wish. This does not alter the input file. Further information on variables is provided in the next section, **BASIC ELEMENTS OF THE AWK LANGUAGE**.

BASIC ELEMENTS OF THE AWK LANGUAGE

This section gives a synopsis of the basic elements of the awk language. You will need these elements in order to define pattern and action pairs.

Comments

You can include comments in an awk program, as in a shell script. A comment begins with the # character and continues till the end of the line.

Constants

There are two types of constant:

.zL
number
string
.zL

number

A number (numeric constant) is a signed or unsigned integer or floating point number. awk does not check its format. If your number contains invalid characters, awk attempts to filter out a valid part and ignores the rest.

integer

An integer is a sequence of digits from 0 to 9.

floating point number

A floating point number consists of a mantissa with or without an exponent.

The mantissa comprises an integer with or without a fractional part. The fractional part is represented by a radix character and an integer.

string

A string (alphanumeric constant) is a sequence of characters, enclosed in double quotes "...". If the double quotes are omitted, awk will interpret the string as a variable name, a number, or an operator.

character

Even a single character must be enclosed in double quotes to prevent awk from interpreting it as the name of a variable. A character is any representable character from the currently valid character set [see ascii(5) and meta-ascii(5)] or one of the following metacharacters, represented as in C:

\"	for	"
\\	for	\
\n	for	newline
\t	for	tab
\b	for	backspace
\r	for	carriage return
\f	for	form feed

Variables

awk allows you to use simple variables and arrays to store values. The special variables are predefined; others can be defined by the user.

Name of a variable

The name of a user-defined variable can be any string made up of underscores (_), uppercase and lowercase letters and digits, beginning with a letter or an underscore. Internal expressions of awk, e.g. control flow statements, must not be used as variable names.

Data type

Variables do not have a data type. You can thus assign either a number or a string to any variable. If the context is clearly numeric, variables are treated as numeric; otherwise, they default to alphanumeric. Numeric variables are converted into alphanumeric variables internally using the format stored in CONVFMT.

Example:

```
x = "Miller";     # Variable x contains the string Miller
x = "3"+4   ;     # Variable x has a value of 7
```

Declaration

awk variables do not need to be explicitly declared. User-defined variables are automatically declared the first time they are used.

Initialization

Special variables are initialized to predefined values by awk. Depending on the context, user-defined variables are initialized by awk to the null string or to 0 by default. If you wish, you can specify other initial values when you call awk.

Note:

- When i>NF, $i will not always be the null string.
- $ variables cannot be initialized on the command line.

Special variables

awk recognizes the special variables listed in the table below. The values awk usually assigns to these variables are indicated in the table. New values may be assigned to the variables by the user.

Variable	Value set by awk
ARGC	Number of elements in the array ARGV
ARGV	Array holding the command line arguments (excluding options and the *prog* argument), numbered from 0 to ARGC-1
CONVFMT	Format for the internal conversion of numbers into character strings (see **FUNCTIONS** below, printf) (default: %%%%.6g, up to 6 places after the decimal point)
ENVIRON	Array holding the values of environment variables, where the indexes are the names of the variables

FILENAME	Name of the current input file, – for standard input
FS	Input field separator (default: any sequence of blanks and tabs)
NF	Number of fields in the current record
NR	Ordinal number of the current record from start of input
FNR	Ordinal number of the current record in the current file
OFS	Output field separator (default: one blank)
ORS	Output record separator (default: newline)
OFMT	Output format for floating point numbers (see **FUNCTIONS** below, printf) (default: %%%%.6g, up to 6 places after the decimal point)
RS	Input record separator (default: newline)

RLENGTH	Length of the string matched by the match function
RSTART	Starting position of the string matched by the match function. Numbering begins with 1. This value always corresponds to the value returned by the match function.
SUBSEP	Subscript string separator for multi-dimensional arrays. The default setting is \034.
$0	Current record
$*n*	Field *n* of the current record
$NF	Last field of the current record

What is the effect of changing special variables?

Example:

The assignment

```
$1 = "new";
```

assigns the string new to $1; but this does not actually alter the first field of the current input record.

This also applies to the following awk settings relating to the input file:

1. The current input file does not change when you assign a new name to FILENAME.
2. When you assign a value to a variable $i where i>NF, NF is assigned the value i.
3. If you assign a new value to NR, you only alter the number assigned to the current line; you do not move to a different line.

   Example:

   The contents of $0 remain the same even if NR is modified:

   ```
   {print NR, $0; NR=NR+34; print NR, $0}
   ```

   A typical output would then be:

   ```
   10 This is the tenth line
   44 This is the tenth line
   ```

Caution:

When you assign a new value to a variable, its old value is deleted. Thus, if you change NF, for example, the information on the number of fields in the current record is lost.

Peculiarity of $ variables:

You can specify the number of a $ variable as a constant or as an expression which evaluates to the number.

Example:

You can use

```
$(NF-1)
```

to access the second-last field.

Array

An array is a set of constants or variables.

An array element is addressed as follows:

array_name[*index*]

array_name
: Name of a variable.

index
: A simple variable.

  The index may be numeric or alphanumeric. The *index* you specify can therefore be a number, a string, or an expression that evaluates to an index value.

`awk` provides two special types of arrays:

- Dynamic arrays

  Arrays, like simple variables, do not need to be declared. Above all, there is no need to define dimensions. New array elements are created automatically as and when required.

- Associative arrays

  Individual array elements can be accessed via an alphanumeric index.

  A special control flow statement is provided in order to process all elements of an associative array:

  `for`(*index* `in` *array*) *statement*

  index assumes the index values present to this point in **random** order, and the specified *statement* is executed once for each array element (see **CONTROL FLOW STATEMENT** below, `for`).

Example:

A file called `expenses` contains various expenses incurred. For each item of expenditure the file shows the date, month, amount, and a brief description, with a colon to separate them. For example:

```
01:January:   40.78:Supplies
05:January: 6789.00:Laser printer
23:March:    240.32:Lamps
11:January:  478.00:Chairs
01:February:  45.00:Journals
```

Using an associative array you can easily calculate total expenditure for each month from the data in this file. The program in the example uses an array called `mexpenses` and the names of the months as an alphanumeric index. For each line, the expenses in the third field (`$3`) are summed up to produce total expenditure for each month appearing in the second field (`$2`).

```
$ awk 'BEGIN {FS=":"}
>      {mexpenses[$2] += $3;}
>      END {for (i in mexpenses) print "Total spent in",
>           i, mexpenses[i]  } ' expenses
Total spent in January 7307.78
Total spent in February 45
Total spent in March 240.32
```

Expressions

An expression can be any of the following:

- *constant*
- *variable*
- *function_call*
- *un_op expression*
- *expression bin_op expression*

- (*expression*)
- *expression* ? *expression* : *expression*
- Numeric or alphanumeric constant (see **BASIC ELEMENTS OF THE AWK LANGUAGE** above).
- Variable (see **BASIC ELEMENTS OF THE AWK LANGUAGE** above).
- Invocation of a predefined function (see **FUNCTIONS** below).
- Expression.
- Unary operator (see the table of operators below).
- Binary operator (see the table of operators below).

Expressions are evaluated and return a value. They may appear both in patterns and in actions.

awk operators

`awk` recognizes all C operators plus the operators for pattern matching and string concatenation. The following table lists all `awk` operators in ascending order of precedence. Operators in the same row have the same precedence.

Warning:

The precedences have changed since previous versions (e.g. "!"). Check existing `awk` programs for ambiguous instances.

Expression Operators

Operation	Operators	Example	Meaning of Example
assignment	`= += -= *= /= %= ^=`	`x *= 2`	`x = x * 2`
conditional	`?:`	`x?y:z`	if `x` is true then `y` else `z`
logical OR	`||`	`y || y`	1 if `x` or `y` is true, 0 otherwise
logical AND	`&&`	`x && y`	1 if `x` and `y` are true, 0 otherwise
array membership	`in`	`i in a`	1 if `a[i]` exists, 0 otherwise
matching	`~ !~`	`$1 ~ /x/`	1 if the first field contains an `x`, 0 otherwise
relational	`< <= == != >= >`	`x == y`	1 if `x` is equal to `y`, 0 otherwise
concatenation		`"a" "bc"`	`"abc"`; there is no explicit concatenation operator
add, subtract	`+ -`	`x + y`	sum of `x` and `y`
multiply, divide, mod	`* / %`	`x % y`	remainder of `x` divided by `y`
unary plus and minus	`+ -`	`-x`	negated value of `x`
logical NOT	`!`	`!$1`	1 if `$1` is zero or null, 0 otherwise
exponentiation	`^ **`	`x ^ y`	x^y
increment, decrement	`++ --`	`++x, x++`	add 1 to `x`
field	`$`	`$i+1`	value of `i`-th field, plus 1
grouping	`( )`	`($i)++`	add 1 to the value of `i`-th field

Evaluation of expressions

Since no data type is prescribed for the operands, you can freely mix numeric and alphanumeric constants. `awk` determines from the context whether a numeric or alphanumeric operation is required.

Please note that, as in C, there are no special truth values. Like C, `awk` treats a value of 0 as false and a non-zero value as true. This means that any non-zero value as an argument of a logical operation is held to be true. If the result of a logical operation is true, it is represented as 1.

Example:

```
$4 ~ /Asia/
```

is 1 if the fourth field of the current line contains `Asia` as a substring, or 0 if it does not.

Patterns

Patterns (selection criteria) are specified by the user as a means of indicating which data is to be selected from the input files. A pattern can have any of the following forms:

- */regexp/*
- *relexp*
- *matchexp*
- *pattern_range*
- *compound_pattern*
- Regular expression.

  `awk` supports extended regular expressions [see `expressions(5)`]. A regular expression is enclosed in slashes /.../.

  Warning:

  Previous versions of `awk` had a special syntax for repeats (`\{m,n\}`). Existing `awk` scripts should therefore be checked.

  Example:

  A regular expression matching any number of occurrences of a, b or c.

  ```
  /[abc]+/
  ```

- *relexp* is an expression (see **Expressions** above) featuring relational operators. The operators and their meanings are:

a	>	*b*	*a* greater than *b*
a	>=	*b*	*a* greater than or equal to *b*
a	<	*b*	*a* less than *b*
a	<=	*b*	*a* less than or equal to *b*
a	==	*b*	*a* equal to *b*
a	!=	*b*	*a* not equal to *b*

  Operands *a* and *b* are any expressions. If both operands are numeric, the comparison is numeric; if not, it is alphanumeric.

- *matchexp* is an expression (see **Expressions** above) featuring pattern matching operators. It involves the comparison of a regular expression (pattern) with a string. The pattern matching operators and their meanings are:

  str ~ p
  : string *str* must match pattern *p*

  str !~ p
  : string *str* must not match pattern *p*

  Using *matchexp* as a pattern allows you to select individual fields.

  Example:

Select all records with a first field starting with A or a:

```
$1 ~ /^[Aa]/
```

The regular expression `^[Aa]` represents strings that begin with A or a. The first field of the record (`$1`) must match (`~`) the regular expression, i.e. begin with A or a.

- A pattern range takes the form:

  /regexp/, /regexp/

  Specifying a range causes the associated action to be executed for all records that lie within the range. The limits of the range (start and end) are defined by two regular expressions. The range begins with the first record containing a string that matches the first regular expression and ends with the first record containing a string that matches the second regular expression.

  Example:

  Select the range from the first line beginning with C to the first line beginning with K and output the first field of every line in the selected range.

  ```
  /^C/, /^K/ {print $1}
  ```

- Logical operators (see **Expressions** above) can be used to negate patterns and to combine several of them to form a single pattern. The logical operators and their meanings are:

  `!`*pat*
  : Negation of pattern *pat*

  pat1 `||` *pat2*
  : *pat1* or *pat2*. The criterion is satisfied if *pat1* or *pat2* matches.

  pat1 `&&` *pat2*
  : *pat1* and *pat2*. The criterion is satisfied if both *pat1* and *pat2* match.

  `(`*pat*`)`
  : Parentheses.

  A compound condition is evaluated from left to right.

  Example:

  Match all records that have an even number of fields and a letter between M (inclusive) and Q (exclusive) in the first field.

  ```
  NF%2==0 && $1 >= "M" && $1 < "Q"
  ```

  You can generally combine patterns in several ways in order to make the same selection. Thus, if the currently valid collating sequence defines the range [M-Q] as the uppercase letters M, N, O, P and Q, the above selection could also be made with pattern matching operators:

  ```
  NF%2==0 && $1 ~ /^[MNOP]/
  ```

  Since the first `awk` condition depends on the collating sequence of the currently valid character set, it may not return the same result in every case. The second `awk` line, by contrast, will always select only those records in which the first field begins with the letter M, N, O or P.

Actions

Actions indicate what to do when a pattern is matched. An action will typically involve processing one of the selected files. An action has to begin in the same line as the associated pattern. If this is not possible, the newline character must be escaped with a backslash. Blanks and tabs between the action and the pattern are ignored.

An action comprises one or more statements and must be enclosed in curly braces {□} as shown below:

{*statement* [*;statement*] ...}

Statements

A statement can be any of the following:

- *expression*

- *control_statement*
- An expression is evaluated but is not put to any further use unless *expression* is in the form of an assignment, an increment or a decrement (see **Expressions** above).
- A *control_statement* allows you to control the flow of an `awk` program (see **CONTROL FLOW STATEMENTS** below).

A single statement may be spread over several lines, in which case each line except the last must end with a backslash. The backslash escapes (cancels the effect of) the newline character.

Multiple statements

You can group together a number of statements within one pair of curly braces {□}. Statements are delimited by means of:

- a semicolon ;
- a right brace }
- a newline character.

CONTROL FLOW STATEMENTS

Control flow statements allow you to control the flow of an `awk` program. `awk` recognizes the following control flow statements:

Statement	Meaning
`break`	terminate a loop
`continue`	skip remainder of loop
`exit`	terminate the `awk` program
`for`	loop counter and looping an array
`if`	conditional statement
`next`	skip to the next input record
`while`	execute iteratively
`do`	execute iteratively
`delete` *array*[*i*]	delete element *i* of the named *array*
`return` *x*	return from a function with a value
`return`	return from a function without a value

The control flow statements are described below in alphabetical order.

break – Terminate a loop

`break` can be used in the body of a `for`, `while`, or `do` loop. `break` causes an immediate exit from the enclosing loop.

`break`

Example:

While records continue to start with a dot, keep reading in the next record. Terminate the loop if the second field of the retrieved record is greater than 1000.

```
{ while($1 ~ /^\./)
    {
       getline;
       if($2 > 1000) break;
    }
}
```

continue – **Skip remainder of loop**

`continue` can be used in the body of a `for`, `while` or `do` loop. The `continue` statement causes the current iteration to be terminated and the next one to begin.

`continue`

Example:

Print even fields only:

```
{
   i=1;
   while(i++ <= NF)
      {
        if(i%2) continue;
        else print $i
      }
}
```

delete – **Delete an array element**

`delete` *array*`[`*i*`]`

`delete` can be used to delete all elements from an array.

Example:

This loop removes all the elements from the array `max`:

```
  for (i in max)
     delete max[i]
```

do – **Execute iteratively**

The statement in a `do` loop (or a `do-while` loop) is executed iteratively while a specified condition continues to be satisfied. In contrast to the `while` loop, the statement in a `do` loop is always executed at least once.

`do {`*statement*`} while (`*expression*`)`

statement
: Statement that is executed in each iteration of the loop. If several statements are to be executed, they have to be grouped together in curly braces {□}.

expression
: Expression (see **Expressions** above) that specifies the condition.

Example:

Print out the individual fields of a record:

```
{ i=0; do {print $(++i)} while (i != NF) }
```

exit – **Terminate the awk program**

`exit` terminates the `awk` program.

If an `END` section is present, `awk` executes the action specified in it; if not, the program is terminated immediately.

`exit`

Example:

If the commercial at symbol `@` appears in the input, print the result and terminate processing:

```
...
/@/ {exit}
...
END {print result}
```

for – **Loop counter**

The statement in a `for` loop is executed iteratively while a condition continues to be satisfied.

`for(`*expr1*`;` *expr2*`;` *expr3*`)` *statement*

expr1

Expression (see **Expressions** above).

expr1 is evaluated once at the start of the `for` statement. *expr1* is often used to initialize incrementing variables.

Example:

```
i=1
```

expr2

Expression (see **Expressions** above).

expr2 is evaluated before each iteration. The specified *statement* is executed only if *expr2* is non-zero (true); otherwise, the loop is terminated.

Example:

```
i<10
```

expr3

Expression (see **Expressions** above).

expr3 is evaluated after each iteration. When incrementing variables are used, *expr3* increments the variable.

Example:

```
i++
```

statement

Statement that is executed in each iteration of the loop. If several statements are to be executed, they have to be grouped together in curly braces {□}.

Example:

Print out the fields of the current record in reverse order:

```
{for(i=NF; i>0; i--) print $i}
```

for – **Looping an array**

This variant of the `for` statement is a special `awk` facility for the handling of arrays.

`for(`*index* `in` *array*`)` *statement*

index

Variable (see **BASIC ELEMENTS OF THE AWK LANGUAGE** above) that assumes all values of the elements of *array* in random order. The index can be numeric or alphanumeric.

array

Array to be processed.

statement

Statement to be executed for each array element. If several statements are to be executed, they have to be grouped together in curly braces {□}.

Example:

The array named `month` contains the number of days in each month. Each array element is subscripted with the name of the month, e.g.

```
month["January"]=31.
```

The following `awk` program prints the name of each month together with the number of days in it.

```
$ awk ' BEGIN { month["January"]=31;
>               month["February"]=28;
>               month["March"]=31;
>               month["April"]=30;
>               month["May"]=31;
>               month["June"]=30;
>               month["July"]=31;
>               month["August"]=31  }
>       END { for(i in month) print i,"has",month[i],"days" } '
May has 31 days
August has 31 days
July has 31 days
April has 30 days
June has 30 days
January has 31 days
March has 31 days
February has 28 days
```

if – **Conditional statement**

The statement in an `if` construct is executed if the specified condition is satisfied.

`if(`*expr*`)` *statement1* [`else` *statement2*]

expr Expression (see **Expressions** above) that defines the condition to be satisfied. If *expr* is non-zero (true), *statement1* is executed.

statement1
Statement to be executed if *expr* is true. If several statements are to be executed, they have to be grouped together in curly braces {□}.

statement2
Statement to be executed if *expr* is false. If several statements are to be executed, they have to be grouped together in curly braces {□}.

Example:

If field 1 is greater than field 2, fields 2 and 3 are printed; if not, fields 4 and 5 are printed:

```
{ if($1 > 2) print $2, $3; else print $4, $5 }
```

next – **Skip to the next input record**

The `next` statement causes `awk` to suspend processing of the current record; statements that follow `next` are not applied to the current record. `awk` then reads the next input record. `NR`, `NF`, `FNR`, `$0`, and `$1` to `$NF` are reset.

Difference between `next` and the `getline` function:

`getline` sets the current record to the next one. Statements that follow `getline` are executed using the next record's values for the `$` variables and for `NR`, `NF`, and `FNR`.

`next`

Example:

Records that begin with a dot are ignored:

```
{ if ($1 ~ /^\./) next }
```

return – **Return from a function**

The body of a function definition may contain a `return` statement that returns control and perhaps a value to the caller.

`return` [*expression*]

Example:

This function computes the maximum of its arguments:

```
function max(m, n) {
   return m > n ? m : n
}
```

The variables `m` and `n` belong to the function `max`; they are unrelated to any other variables of the same names elsewhere in the program.

while – Execute iteratively

The statement in a `while` loop is executed iteratively while a specified condition continues to be satisfied.

`while(`*expr*`)` *statement*

expr Expression (see **Expressions** above) that specifies the condition.

statement
Statement that is executed in each iteration of the loop. If several statements are to be executed, they have to be grouped together in curly braces {□}.

Example:

Print all input fields, writing each field in a separate output line:

```
{ i = 1;
  while (i <= NF) {
      print $i
      i++
  }
}
```

FUNCTIONS

`awk` provides a wide range of built-in functions and also offers you the option of defining functions of your own:

`function` *name*`(`*arg*, ...`)` {*statements*}

The {*statements*} may be preceded by a newline character. There may also be blank lines within the curly braces {□}. A function definition has the same precedence as *pattern* {*action*} pairs in the main section of an `awk` program.

Within an action section, function calls can be entered anywhere in an expression, except before the function declaration. There must be no space between the function name and the left parenthesis when a function is called. Nested and recursive function calls are legal.

Though most functions do not require you to enclose arguments in parentheses, it is good practice to use them as a means of increasing program transparency. When you pass an array as an argument, a pointer to the array is passed (call by reference), which means that you can change the elements of the array from the function. In the case of scalar variables, the value of the variable is copied and passed (call by value), which means that you cannot change the value of the variable from the function. The scope of function arguments is restricted to the local function, whereas the scope of all other variables is always global. If you need a local variable in a function, define it at the end of the argument list in the function definition. Any variable in the argument list for which no current argument exists is a local variable with a predefined value of `0`.

As in C, some functions return a result (e.g. *exp*), while others are procedural in character (e.g. output functions). The `return` statement can be used with or without a return value or may be omitted entirely. In the latter case, the return value would be undefined if it were to be accessed.

Example:

In the example below, the function named `search` looks for the string `who` in the array `allnames` and returns the index or `–1`. The third argument, *incr*, is used as a local variable.

```
   ...
{ print $1, search($1, allnames) }
   ...
function search(who, allnames, incr)
```

```
{
   for (incr=0; allnames[incr]; incr++)
      if (index(allnames[incr], who) == 1
          && length(allnames[incr]) == length(who))
             return incr
   return -1
}
```

Built-in functions

Function	Purpose
Input function	
`getline`	Read input record
Output functions	
`print([`*arg*`, ...])`	Standard output function
`printf(`*format* `[`*arg*`, ...])`	Formatted output
Arithmetic functions	
`atan2(`*y*`,`*x*`)`	Arc tangent of *y*/*x*
`cos(`*x*`)`	Cosine
`exp(`*x*`)`	Exponential function
`int(`*x*`)`	Truncate to integer
`log(`*x*`)`	Natural logarithm
`rand()`	Return a random number
`sin(`*x*`)`	Sine
`sqrt(`*x*`)`	Square root
`srand([`*x*`])`	Set the seed (initial value) for `rand()`

Function	Purpose
String functions	
`gsub(`*RE*`,`*repl*`[,`*instr*`])`	Global substitution function
`index(`*str1*`,`*str2*`)`	Return first occurrence of substring
`length([`*str*`])`	Return length of string
`match(`*str*`,`*RE*`)`	Check whether string *str* matches regular expression *RE*
`split(`*str*`,`*array*`, [`*sep*`])`	Subdivide string
`sprintf(`*format*`,`*e1*`,`*e2*`,...)`	Return formatted output as string
`sub(`*RE*`, `*repl*`[,`*instr*`])`	Substitution function
`substr(`*str*`,`*m*`,[`*n*`])`	Define substring
`tolower(`*zk*`)`	Conversion to lowercase letters
`toupper(`*zk*`)`	Conversion to uppercase letters
General functions	

close(*expr*)	Close file or pipe
system(*expr*)	Call shell command

The following section describes each of these functions in alphabetical order together with the associated arguments. The argument you specify can either be a constant or an expression (see **Expressions** above). awk first evaluates the expression arguments and then applies the function to the computed results.

atan2 – **Arc tangent**

atan2 calculates the arc tangent of the quotient of two numbers. atan2(*y*,*x*) returns the arc tangent of *y*/*x*.

atan2(*y*,*x*)

y,*x* Numbers that produce the quotient for which the arc tangent is to be calculated.

close – **Close file or pipe**

close closes the specified file or pipe.

close(*expr*)

expr Name of the file or pipe to be closed (see the section on "Redirecting output" under the descriptions of the print and printf functions).

cos – **Cosine**

cos calculates the cosine of a number.

cos(*x*)

x Number for which the cosine is to be calculated.

exp – **Exponential function**

exp calculates e to the power of *x*.

exp(*x*)

x Number for which ex is to be computed.

getline – **Read a record**

awk retrieves a record as directed (see also the control flow statement next).

getline has several different formats, with the following return values:

1 successful execution

0 end-of-file

–1 error

getline

awk reads the next input record from the input file into $0. NR, NF, FNR, $0, and $1 to $NF are reset.

Example:

If a record contains %%%, the next record is read. In other words, input records containing %%% are ignored.

```
/%%%/ {getline}
```

getline < *file*

awk reads a record from the named *file* into $0. NF, $0, and $1 to $NF are reset.

file Name of the file from which a record is to be read.

getline *var*

awk fetches the next input record from the input file and puts it into the variable *var*. NR and FNR are reset.

var Variable into which the next record is to be read.

getline *var* < *file*

awk fetches a record from the named *file* and puts it into the variable *var*. NR, NF, FNR, $0, and $1 to $NF

remain unchanged.

var Variable into which the record is to be read.

file Name of the file from which the record is to be read.

command | `getline` [*var*]

The output of the named *command* is redirected to `getline`. Each `getline` call in this format causes `awk` to read the next line from the output of *command* and write it into `$0` or the variable *var*.

If *var* is specified, `NR`, `NF`, `FNR`, `$0`, and `$1` to `$NF` remain unchanged; if not, `NF`, `$0`, and `$1` to `$NF` are reset.

This construct is equivalent to calling the C function `popen(3S)` with mode `r`.

var Variable into which the record is to be written.

var not specified:

The record is written into `$0`.

command

Name of the command whose output is to be read.

gsub – Global substitution function

`gsub` globally substitutes the string *repl* for all strings in `$0` or *instr* that match the extended regular expression *RE*.

`gsub` returns the number of substitutions.

`gsub(`*RE*,*repl*[,*instr*]`)`

RE Extended regular expression that specifies the pattern to be matched.

repl String to be substituted for the strings that match *RE*.

instr

String in which the substitution is to be made.

instr not specified:

Substitution is done in `$0`.

index – Search for substrings

`index` searches for a substring within a string. If the substring is present, `index` returns the starting character position (numbered from 1 onward) of its first occurrence in the string; if not, it returns a value of 0.

`index(`*str1*,*str2*`)`

str1 String in which `index` looks for the substring.

str2 Substring that `index` looks for.

Example:

```
index("PaPa-MaMa","Pa")
```

returns 1.

int – Truncate to integer

`int` returns the largest integer equal to or smaller than the argument.

`int(`*x*`)`

x Number that is to be truncated to its integer part.

length – Return length

`length` returns the length of a string.

`length[(`*str*`)]`

str `length` returns the length of string *str*.

str not specified:

`length` returns the length of the current input record `$0`.

log – **Logarithm**

`log` calculates the natural base `e` logarithm.

`log(`*x*`)`

x Number whose natural log is to be computed.

match – **Match regular expressions**

`match` checks whether a string in *str* matches the extended regular expression in *RE*.

If a matching string is found, `match` returns the character position in *str* (numbered from 1 onward) at which the string begins; if not, it returns 0.

The variable `RSTART` is set to the return value of `match`; `RLENGTH` is set to the length of the matching string (or –1 if no matching string is found).

`match(`*str*,*RE*`)`

str String in which the pattern is to be matched.

RE Extended regular expression.

print – **Standard output function**

`print` is the standard output function. `print` outputs either the current record or the specified arguments and terminates its output with the output record separator `ORS`. For further details refer to *Output format* below.

`print(`*arg1*[[,]*arg2*]...`)` [*redirection*]

No argument specified:

`print` writes the current input record on standard output.

arg1arg2

Arguments that are to be printed.

`print` evaluates the expression arguments and concatenates the results in the order in which the arguments are specified.

arg1,*arg2*

Arguments that are to be printed. `print` outputs the evaluated expression arguments in the specified order, separated by the output field separator `OFS` if they are separated by commas in the `print` statement.

redirection

Output can be redirected to a file or piped to a program. You can use up to 10 output files.

redirection can be in the form of: >, >>, name of program.

>*file*

The output is written to the named *file*. The former contents of *file* are deleted the first time `print` is called. All subsequent `print` or `printf` outputs to *file* in the same `awk` program are appended to the end of *file*. Unless explicitly closed, *file* remains open until the end of the `awk` program.

>>*file*

The output is appended to the previous contents of *file*. Unless explicitly closed, *file* remains open until the end of the `awk` program.

|*prog*

The output is piped to the program named *prog*.

You are only permitted to open one pipe to *prog* within an `awk` program, but you can pipe any number of `print` or `printf` outputs to it.

This construct is equivalent to calling the C function `popen(3S)` with mode `w`.

Unless explicitly closed, the pipe remains open until the end of the `awk` program.

The file or program name can specified directly (enclosed in "...") or via a variable that evaluates to the filename.

Caution:

If you redirect output to the input file, the input file will be destroyed without any warning.

Output format

`print` outputs integers in decimal and prints strings at full length. Apart from that, the output format is contingent on the following predefined variables:

`OFS` – output field separator
`OFS` is one space by default. If you wish, you can assign any one character to `OFS` to change the output field separator.

`ORS` – output record separator
`ORS` is the newline character by default. If you wish, you can assign any one character to `ORS` to change the output record separator.

`OFMT` – floating point output format
`OFMT` defines the output format for floating point values and is set to `%.6g` by default. This means that the fractional part of a floating point number is printed with a maximum of 6 places. If you wish, you can assign a different `printf` format for floating point numbers to `OFMT` (see `printf` below).

Example 1

Print the first and second fields, separated by a blank:

```
{print $1,$2}
```

Example 2

Concatenate the first and second fields without an output field separator:

```
{print $1$2}
```
or
```
{print $1 $2}
```

printf – Formatted output

`printf` is the output function for formatted output. The output format can be specified as in the standard `printf(3S)` function in C.

`printf` (*format*,*arg*,...) [*redirection*]

format

String defining the output format. The output format comprises plain characters and format elements (conversion specifications). Printable characters are output unaltered. The metacharacters listed in the "Basic elements" section above are converted immediately. For example, `\n` sets the position to the start of the next line.

All format elements begins with the percent sign. Specification of a *digit* after the percent sign defines the number of places. The most common format elements are presented in the following table:

Format element	Meaning
`%c`	single character
`%d`, `%i`	decimal integer
`%e`, `%E`	floating point number in exponential notation, e.g. 5.234e+2
`%f`	floating point number, e.g. 52.34
`%g`, `%G`	`%e` or `%f`, whichever is shorter

`%o`	octal integer (base 8)
`%s`	character string
`%u`	unsigned decimal integer
`%x`, `%X`	hexadecimal integer (base 16)

arg Arguments that are to be printed.

`printf` evaluates the expression arguments, allocates them in the given order to the specifications in `format`, and outputs them in the appropriate format.

- If the format element is incompatible with the argument, e.g. a numeric format specification for an alphanumeric argument, a 0 is printed.
- If there are more arguments than format elements, the excess arguments are ignored, i.e. not printed.
- If there are more format elements than arguments, an error message is issued.

redirection

Redirection is as for `print`.

redirection not specified:

`printf` prints on standard output.

Example:

Field 1 is printed as a decimal number with at least 2 positions, followed by `**` as a separator, followed by field 2 as a string of at least 5 characters, followed by newline:

```
{ printf("%2d**%5s\n", $1,$2) }
```

rand – **Return a random number**

`rand` returns a random number *r*, where 0 ≤ *r* < 1.

`rand`

Also refer to `srand`.

sin – **Sine**

`sin` returns the sine of a number.

`sin(`*x*`)`

x Number whose sine is to be computed.

split – **Subdivide strings**

`split` divides a string into substrings and stores each substring as an element in an array. The elements are subscripted in ascending order, starting with 1.

`split` returns the number of array elements.

`split(`*str*`,`*array*`[,`*sep*`])`

str String that is to be split.

array

Name of the resulting array.

sep Extended regular expression specifying the characters that act as a separator between the substrings in *str*.

sep not specified:

`FS` is used as the separator.

Example:

```
{
     s=split("january:february:march", months, ":");
     for(i=1; i<=s; i++) print months[i];
}
```

produces the output

```
january
february
march
```

sprintf – **Return formatted output as a string**

`sprintf` formats in exactly the same way as `printf`, but there is no direct output. `sprintf` instead returns the formatted output as a string, which could then be assigned to a variable or used for a similar purpose.

`sprintf` (*format*,*arg*,...)

format
: String defining the output format (see `printf`).

arg Arguments that are to be output (see `printf`).

Example:

The following `awk` program fragment produces the same output as the example given under `printf`.

```
{ x = sprintf("%2d**%5s\n", $1,$2); print x }
```

sqrt – **Calculate the square root**

`sqrt` calculates the square root of a number.

`sqrt` (*x*)

x Number whose square root is to be computed.

srand – **Set the seed for the rand function**

`srand` sets the seed (starting point) for the `rand` function to the number *x*, or to the current time if no argument is specified.

`srand` ([*x*])

x Number that is to serve as the seed for `rand`.

sub – **Substitution function**

`sub` substitutes the string *repl* for the first instance of a string in `$0` or *instr* that matches the extended regular expression *RE*.

`sub` returns the number of substitutions.

`sub` (*RE*,*repl*[,*instr*])

RE Extended regular expression that specifies the pattern to be matched.

repl String to be substituted for the strings that match *RE*.

instr
: String in which the substitution is to be made.

instr not specified:

The substitution is done in `$0`.

substr – **Define a substring**

`substr` extracts a substring from a string.

`substr` (*str*,*m*[,*n*])

str String from which the substring is to be extracted.

m Position in *str* at which the substring begins. Character positions are numbered consecutively from left

to right, starting with one.

n Maximum length of the substring.

n not specified:

The substring extends to the end of `str`.

Example:

```
{
   x = substr("060789",3,2); print "Month = "x
}
```

produces the output:

```
Month = 07
```

system – Call shell command

`system` executes the specified shell command and returns its exit status.

`system(`*command*`)`

command

Name of the shell command to be executed.

tolower – Convert to lowercase letters

`tolower` converts all uppercase letters in a character string to lowercase letters.

`tolower(`*zk*`)`

zk Character string to be converted to lowercase letters

toupper – Convert to uppercase letters

`toupper` converts all lowercase letters in a character string to uppercase letters.

`toupper(`*zk*`)`

zk Character string to be converted to uppercase letters

DIAGNOSTICS

If an `awk` program contains errors, `awk` issues corresponding error messages and exits immediately. The error messages indicate the cause of the error, if detectable by `awk`, and the `awk` program line in which `awk` thinks the error is to be found. Typical error messages are:

```
awk: syntax error at source line xxx
awk: illegal statement source line number xxx
```

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In regular bracketed expressions, the `LC_COLLATE` environment variable governs the scope of character ranges, equivalence classes and collating elements, while the `LC_CTYPE` environment variable governs the scope of character classes. `LC_COLLATE` governs the behavior of relational operators in string comparisons. `LC_TYPE` governs the behavior of `toupper`/`tolower` conversions.

`LC_NUMERIC` governs the representation of the radix character, the exponentiation symbol and the digit grouping character for output and internal conversions, though not for the specification of values within an `awk` program or the assignment of variables in arguments.

If `LC_MESSAGES`, `LC_COLLATE`, `LC_CTYPE` or `LC_NUMERIC` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

Output all input lines in which field 3 is greater than field 5:

```
$ awk '$3 > $5' file
```

Since no action has been specified, `awk` prints the selected lines by default.

Example 2

Print every 10th line of a file:

```
$ awk '(NR % 10) == 0' file
```

Example 3

Print the second to last and the last field in each line, separated by a colon:

```
$ awk 'BEGIN {OFS=":"}
>      {print $(NF-1), $NF} ' file
```

If a line consists of a single field, the entire line is output twice, separated by a colon (first `$0`, then `$1`).

Example 4

Add up the values of the first field of every line and print the total and average at the end.

```
$ awk '{s += $1}
>      END {print "Total: ", s, "Average: ", s/NR}'
>      file
```

Example 5

Find a preprocessor `if` directive, i.e. a range of lines in which the first line begins with `#if` and the last line with `#endif`.

```
$ awk '/^#if/, /^#endif/' file
```

Example 6

Print all lines in which the first field differs from that of the previous line:

```
$ awk '$1 != prev { print; prev = $1 } ' file
```

Example 7

`file` contains a list of data about young people, with the second field containing one of the entries `school`, `university`, `apprenticeship` or `elsewhere`. For statistical purposes, you want to count how many are at school and university:

```
$ awk '$2 ~ /school/ {incr["school"]++}
>     $2 ~ /university/ {incr["university"]++}
>     END {print "school:" incr["school"];
>          print "university:" incr["university"]} ' file
```

Example 8

The file `contents` lists the table of contents of a text. The table of contents is organized in decimal classification and has the format:

```
1. Foreword
2. Introduction
3. The Game of Chess
3.1. History
3.2. Rules
3.2.1 Setting Up the Figures
.
.
.
4. The Game of Checkers/Draughts
4.1. History
.
.
```

```
.
8. Index
```

The following `awk` program can be used to give the list a more orderly format:

```
$ awk '{$1=$1"      ";
>      $1=substr($1,1,6);
>      print $0} ' contents >> con.form
```

The output lines are prepared in the following stages:

First, six blanks are added to the end of the first field (`$1=$1"      "`). Then the first field is truncated to six characters. Thus the first field of each line is 6 characters long, and field 2 always starts at column 7. The output in the file `con.form` will be as follows:

```
1.    Foreword
2.    Introduction
3.    The Game of Chess
3.1.  History
3.2.  Rules
3.2.1 Setting Up the Figures
.
.
.
4.    The Game of Checkers/Draughts
4.1.  History
.
.
.
8.    Index
```

Example 9

The following `awk` program in the file `prog` prints the number of fields and the actual fields of each record. The record separator has been redefined as the dollar sign. The field separators are thus blanks, tabs, and the newline character:

```
BEGIN { RS="$"; printf "Record\Num" }
      { printf ("\n%4d3d\", NR, NF);
        for(i=1;i<=NF; i++) printf "%s:", $i }
END {print"\n"}
```

The file `text` contains the following text:

```
first record$  second   record     $
$
fourth     and  last
record$
```

The call:

```
$ awk –f prog text
```

returns:

```
Record  Num
     1    2      first:record:
     2    2      second:record:
     3    0
     4    4      fourth:and:last:record:
     5    0
```

Example 10

You now change the file `text` to:

```
&&
first&&record$second record$$fourth
and&
last
```

```
record&
```

and call `awk` again, this time using the `-F` option to change the field separator to `&`.

```
$ awk -F"&" -f prog text
```

The output returned is:

```
Record  Num
     1    6    :::first::record:
     2    1    second record:
     3    0
     4    8    fourth:and::last::record:::
```

This example illustrates how fields are separated when a non-standard separator is used. The first line (`&&`) of the `text` file is a part of the first record and now yields 3 fields, for example, because each individual separator in a string of separators (`&&`) is counted, and the newline implicitly acts as a separator as well (2 `&` + 1 newline = 3).

SEE ALSO

`egrep`(1), `fgrep`(1), `grep`(1), `lex`(1), `sed`(1), `expressions`(5).

Kernighan, B. W.; Pike, R.: *The UNIX Programming Environment*.

Tare, R. S.: *UNIX Utilities*.

Aho, A.; Kernighan, B. W.; Weinberger, P.: *The AWK Programming Language*.

User's Reference Manual

banner(1)

NAME

`banner` - output arguments in large letters

SYNOPSIS

`banner` *string* ...

DESCRIPTION

`banner` writes argument strings in large letters on standard output as shown in the example, putting each string on a separate line. The font of the output is that of the US-ASCII character set [see `ascii(5)`].

ARGUMENTS

string

string may consist of any alphanumeric characters. If more than one string is entered, the strings must be separated by one or more spaces. Spaces within a string and shell metacharacters [see `specialchar(5)`] must be individually escaped with a backslash \ or enclosed in single '...' or double "..." quotes. The maximum number of characters printed per line is 10. Excess characters are simply ignored. A distinction is made between lowercase and uppercase letters in the string. Lowercase letters are output one line smaller.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Print the string `This is a banner!` in large letters on the screen:

```
$ banner This is a banner!
#######
   #     #   #    #     ####
   #     #   #    #    #
   #     #####    #     ####
   #     #   #    #         #
   #     #   #    #    #    #
   #     #   #    #     ####
#     ####
#    #
#     ####
#         #
#    #    #
#     ####
  ##
 #  #
#    #
######
#    #
#    #
######                                          ###
#     #    ##     #    #  #    #  ###### #####   ###
#     #   #  #    ##   #  ##   #  #      #    #  ###
######   #    #   # #  #  # #  #  #####  #    #   #
```

```
#     #  ######   #  # #  #  # #  #      #####
#     #  #    #   #   ##  #   ##  #      #   #   ###
######   #    #   #    #  #    #  ###### #    #  ###
```

SEE ALSO

echo(1) .

User's Reference Manual

basename(1)

NAME

`basename` - strip file name affixes

SYNOPSIS

`basename` *string* [*suffix*]

DESCRIPTION

You can use `basename` to

- extract the basic file name (basename) from the full path name,
- strip any suffixes from the file name.

`basename` strips all characters up to and including the last / from the specified string and writes the result to standard output. The basic file name can thus be separated from its path prefix. If you also specify a string suffix as a command-line argument, `basename` will strip this suffix as well. `basename` is useful within backquotes `...` in shell scripts [see `sh(1)`].

OPERANDS

string

string can be any character string.

`basename` deletes all characters up to and including the last / from *string* and writes the result to standard output. Strings that do not include a slash are output unmodified.

suffix

suffix can be any character string.

If the specified *suffix* matches the end of *string*, *string* is output without *suffix*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

The name `prog` is to be generated from `/home/catherine/program`:

```
$ basename /home/catherine/program ram
prog
```

Example 2

The following shell script compiles a C source program. `basename` generates the name of the compiled program from the file name used as a command-line argument for the shell script. The compiled program is stored in an executable file in the current working directory. The shell script is called `compile`.

Contents of `compile`:

```
cc –o `basename $1 .c` $1
```

If you call *compile* as follows:

```
$ compile /home/anna/cprogs/tab.c
```

the file name of the C source file is passed to the `cc` command [control program for compiling and linking C programs, see `cc`(1)] as positional parameter `$1` [see `sh(1)`]. The shell substitutes the output of `basename`

for the operand of `cc`'s `-o` option. The name of the executable file is `tab`.

SEE ALSO

`dirname`(1), `ed`(1), `sh`(1).

User's Reference Manual

basename(1-ucb)
BSD Compatibility Package

NAME

`basename` - display portions of pathnames

SYNOPSIS

`/usr/ucb/basename` *string* [*suffix*]

DESCRIPTION

`basename` deletes any prefix ending in "`/`" and the *suffix*, if present in *string*. It directs the result to the standard output, and is normally used inside substitution marks (`` ` `` `` ` ``) within shell procedures. The *suffix* is a pattern as defined on the `ed`(1) manual page.

EXAMPLE

This shell procedure invoked with the argument `/usr/src/bin/cat.c` compiles the named file and moves the output to `cat` in the current directory:

```
cc $1
mv a.out `basename $1 .c`
```

SEE ALSO

`ed`(1), `sh`(1).

User's Reference Manual

batch(1)

NAME

`batch -` execute commands at a later time

SYNOPSIS

`batch` <RETURN> □
command ... <RETURN> □
<CTRL-D>

DESCRIPTION

`batch` reads commands from standard input, puts them in a queue, and executes them when system load level permits.

Standard output and standard error output of the commands to be executed are sent to the user by `mail` unless they are redirected elsewhere. The environment variables, the current directory, the permissions for new files [see `umask(1)`] and the maximum permissible file size [see `ulimit(1)`] are retained, but open files and priorities are lost, and the `trap(1)` command (shell built-in for catching signals) is deactivated.

`batch` writes the job number and the schedule time to standard error.

Jobs scheduled with `batch` are run even if the user who scheduled them logs out of the system.

`batch` has exactly the same effect as `at -qb` with no further options.

Before the call

If the file `/etc/cron.d/at.allow` exists, you can only use `batch` if your login name appears in it.

If the file `/etc/cron.d/at.allow` does not exist, you can only use `batch` if your login name does **not** appear in the file `/etc/cron.d/at.deny`.

If neither `/etc/cron.d/at.allow` nor `/etc/cron.d/at.deny` exists, only the system administrator is allowed to use `batch`.

If only an empty deny file exists, for example, everyone is allowed to use `batch`.

Only the system administrator is allowed to create and modify the allow and deny files. Each line in these files contains precisely one login name.

OPERANDS

command

Any command or shell script. You can specify more than one *command* at a time, using semicolons or newlines to separate them. The resulting command list is executed under a single job number.

ERROR MESSAGES

`at: you are not authorized to use at. Sorry.`

Permission to use `batch` denied (see **Before the call**).

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_TIME` environment variable governs the format of the date and time specifications.

If `LC_MESSAGES` or `LC_TIME` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables were set.

The `LC_ALL` environment variable governs the entire internationalized environment. `LC_ALL` takes precedence over all other environment variables in the area of internationalization.

EXAMPLES

In the following example, the standard input is redirected, and `batch` takes its work from the file `jobs`:

```
$ batch < jobs □
job 604763316.b at Wed Mar 1 14:48:36 1989
```

The jobs contained in the file `jobs` are run in sequential order as background processes by `batch`. When the job is complete, you can have the result displayed on the screen by `mail`.

FILES

`/etc/cron.d/at.allow`
: List of login names with permission to use `batch`. One login name is entered per line.

`/etc/cron.d/at.deny`
: List of login names explicitly denied permission to use `batch`. One login name is entered per line.

`/var/spool/cron/atjobs`
: Directory containing a separate file for each `batch` job which has not yet been executed. Each `batch` job is allocated a file of its own with the filename *jobnumber*`.b`.

`/etc/cron.d/queuedefs`
: File containing scheduling information.

SEE ALSO

`at`(1), `crontab`(1), `mail`(1), `ulimit`(1), `ulimit`(2), `queuedefs`(4).

User's Reference Manual

bc(1)

NAME

`bc` - arbitrary precision arithmetic language

SYNOPSIS

`bc` [`-c`] [`-d`] [`-l`] [`--`] [*file*]...

DESCRIPTION

You can use `bc` to perform arithmetical calculations. `bc` is an interactive program for a C-like input language and is actually a preprocessor for `dc`(1) . `bc` automatically invokes the desk calculator `dc` and outputs results with any required degree of precision.

OPTIONS

`-c`
`-d`

`bc` simply generates statements for the desk calculator `dc` and writes them to standard output, without actually invoking `dc`. The `-d` option has the same effect as the `-c` option.

`-l` `-l` stands for `/usr/lib/lib.b`, which is a library containing `bc` programs for various mathematical functions.

The `-l` option must be specified if you want to use any of the following functions:

`s`(*x*) sine

`c`(*x*) cosine

`e`(*x*) base e exponential function

`l`(*x*) natural logarithm

`a`(*x*) arctangent

`j`(*n*,*x*)
n-th order Bessel function

`--` If *file* begins with a dash (-), the end of the command line options must be marked with `--`.

file Name of file containing a `bc` program. You may specify more than one file. When all statements from all files have been processed, `bc` reads from standard input. You can then enter further statements.

file not specified: `bc` reads from standard input.

ELEMENTS OF BC PROGRAMS

A`bc` program consists of

- definitions
- statements
- comments

The following symbols are used in defining the structure of `bc` programs:

L stands for one of the letters a-z

E stands for an expression

S stands for a statement

Comments

Comments are enclosed in /* ... */ as in C.

Statements

Statements in `bc` can be:

- Expressions (see **Expressions**)

  The value of a statement that is an expression is printed unless the main operator is an assignment.
- Blocks (grouped statements): {*S*; ...; *S*}
- Conditional statement: `if` (*E*) *S*

  If expression *E* is true, i.e. has a non-zero value, then statement *S* is executed.
- Iteration statements:
  - `while` (*E*) *S*

    Expression *E* is evaluated and if it has a non-zero value, statement *S* is executed. *E* is then evaluated again, and *S* is executed again if *E* is still non-zero. This process is repeated for as long as *E* has a non-zero value.
  - `for` (*E*; *E*; *E*) *S*

    First, the first expression is evaluated. Next, the second expression is evaluated and if it has a non-zero value, statement *S* is executed. Lastly, the third expression is evaluated. Then the second expression is evaluated again and statement *S* is again executed if it is non-zero, and so on. Unlike in C programs, a for statement must always contain three expressions.
- Jump statement: break

  The break statement can be used only within an iteration statement. It causes termination of the nearest while or for statement. Program execution continues with the statement that follows the terminated iteration statement.
- Termination statement: quit

  The quit statement stops execution of a `bc` program. The quit statement is interpreted as soon as it is read, not when the `bc` program is executed.

  Example:

  The following `bc` program terminates immediately without printing the value of *a*:

```
a=5
if (a>10) quit
(a)
```

You can separate statements from one another with a semicolon or a newline character.

Expressions

Expressions consist of operands and operators.

Operands are names or arbitrarily long numbers with optional sign and decimal point.

Names

L simple variables

L function names

L[*E*] array elements

`ibase`
: base (radix) for input numbers, default: 10

`obase`
: base (radix) for output numbers, default: 10

`scale`
: number of fractional digits, default: 0

If arrays are used as function arguments or defined as automatic variables, empty square brackets must follow the array name.

The same name may be used simultaneously for an array, a function, and a simple variable. All variables are

global to a `bc` program.

Other operands

(*E*) result of *E*

`sqrt`(*E*)
: square root of *E*

`length`(*E*)
: number of significant decimal digits in *E*

`scale`(*E*)
: number of fractional digits in *E*

L(*E*, ...,*E*)

Operators

`+ − * /`
: addition, subtraction, multiplication, division

`^` power operator

`%` remainder of integer division (can now also be applied to floating-point numbers.

`++ −−`
: increment and decrement operators, which can be applied to names in prefix or postfix notation

`< <= == >= > !=`
: relational operators (less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to)

`=` assignment operator

`=@` compound assignment operators, where `a=@b` is the same as `a=a@b`. `@` can be any of the operators `+`, `−`, `*`, `/`, `^`, or `%`.

The logical operators `&&` and `||` are **not** recognized by the `bc` command.

Function definitions

```
define L (L, ...,L) {
     auto L, ...,L
     S; ...;S
     return (E)
 }
```

Example:

```
   define p(x) {
         auto q
         q = p * p
         return (q)
   }
```

Declaring the identifiers of a function as `auto` restricts their scope to that function. All function arguments are passed by value.

Functions in the math library /usr/lib/lib.b

Definitions of the mathematical functions listed below are contained in the library `/usr/lib/lib.b`. The functions can be accessed by calling `bc` with option `−l`.

`s`(*x*) sine

`c`(*x*) cosine

`e`(*x*) base e exponential function

`l`(*x*) natural logarithm

`a`(*x*) arctangent

`j(`*n*`,`*x*`)`
: *n*-th order Bessel function

x values in mathematical functions must be specified in absolute radian measure.

If you have write permission for `/usr/lib/lib.b`, you can add definitions of further functions and also modify or delete existing ones.

Defining bases for input and output numbers

With `ibase` and `obase` you can specify the base used for interpreting input and output values (input and output number radix). The following rules apply:

1. If you do not explicitly assign values to `ibase` and `obase`, input numbers are interpreted as decimal and results are output in decimal.
2. If you have already defined the input base with an `ibase=`*n* statement, the number that you use to define the output base in an `obase=`*m* statement must be in input base *n* as well.

Example:

The input base is to be 2, the output base 16:

```
$ bc
ibase=2
obase=10000
10100000/1010
10
```

Fractional digits

Each expression *E* in `bc` is associated with a specific number of fractional digits. You can use `scale` to inspect or change this number and the `scale(`*E*`)` function just to inspect it.

Example:

In the following example the value of operand *a* is divided by the value of operand *b* with `scale` initially left unset: the result contains no fractional digits. Then `scale` is assigned a value of 8: the result of the division is now correct to 8 places after the decimal point.

Finally we inspect the number of fractional digits in the result and the value of `scale`.

```
$ bc
a=15.0
b=7.8
a/b
1
scale=8
a/b
1.92307692
scale(a/b)
8
scale
8
```

If you join two expressions using an operator, the number of fractional digits associated with the result is governed by a rule specific to the operator you use. The rules for `bc` operators are described below. Various symbols are used in the descriptions:

a =first operand

b =second operand

R =number of fractional digits in the result of a calculation

A =`scale(`*a*`)`

B =`scale(`*b*`)`

–
++
––

The unary minus sign and the increment and decrement operators ++ and –– (in prefix and postfix notation) do not affect the number of fractional digits.

Rule: `scale`(*E*) = `scale`(–*E*) = `scale`(––*E*) = `scale`(++*E*)...

Example:

a is assigned a value with three fractional digits. The query function `scale`(*a*) here always returns 3, regardless of whether *a* has a –, –– or ++ operator and regardless of the fact that `scale` was previously assigned a different value:

```
$ bc
scale=1
a=1.123
scale(a)
3
scale(-a)
-3
scale(a++)
3
scale(--a)
3
```

+
–

With the binary operators + and –, *R* is equal to the number of fractional digits in the operand with the most fractional digits, regardless of whether you have previously assigned `scale` some other value.

Rule: *R* = *max*(*A*,*B*)

Example:

`scale` is assigned a value of 1. Operand *a* is assigned a value with 2 fractional digits, *b* a value with three fractional digits. Thus *b* has more fractional digits than *a*, and also more than `scale`. The query function returns 3 for both operands, the greater number in *b* taking precedence.

```
$ bc
scale=1
a=0.12
b=0.123
scale(a+b)
3
scale(b-a)
3
```

*

With multiplications, a value previously assigned to `scale` is significant: `bc` first calculates *max*, which is the highest of the values `scale`, *A* and *B*. It then forms the sum of *A* and *B* and compares this value with *max*. *R* is then the lower (*min*) of these two values.

Rule: *R* = *min*(*A*+*B*, *max*(`scale`, *A*, *B*))

Example:

`scale` has a value of 9, *A* and *B* are both 1. Thus the highest of the three values is 9. The sum of the number of fractional digits in the two operands is 2. The number of fractional digits in the result of the multiplication is the lower from the comparison of *max* and this sum, which is 2.

```
$ bc
scale=9
a=0.1
b=0.1
scale(a*b)
```

2

/ With divisions, the precision of the result is equal to the value of scale:

Rule: R = `scale`

Example:

`scale` is first given a value of 8. Then the operands are assigned integer values. The result is also an integer. In spite of that, the query function returns 8, and the result is shown correct to 8 places after the decimal point.

```
$ bc
scale=8
a=16
b=4
scale(a/b)
8
a/b
4.00000000
```

^ With the power operator, R is formed as follows:

- If the integer exponent `e` is equal to or greater than 0:

  `bc` takes the higher (*max*) of the two values `scale` and A. It then multiplies A by the absolute value m of the exponent, compares the result with *max*, and takes the lower of the two values.

  Rule: R = *min*(A*m, *max*(`scale`, A))

- If the integer exponent `e` is less than 0:

  The precision of the result is equal to the value of `scale`.

  Rule: R = `scale`

Example 1

`scale` is given a value of 7. a has one fractional digit, and the absolute value of exponent e is 4, i.e. greater than 0. The higher value from the comparison of `scale` and a is 7. However, the result of multiplying a and m is 4. Thus the number of fractional digits after exponentiation is 4:

```
$ bc
scale=7
a=3.1
e=4
scale(a^e)
4
a^e
92.3512
```

Example 2

However, if you set the exponent e to –4, the number of fractional digits is equal to the value of `scale`:

```
$ bc
scale=7
a=3.1
e=-4
scale(a^e)
7
a^e
.0108281
```

=
=@

With the assignment operators, the value of R is equal to that of A after assignment. For a given compound operator `=@`, the rule for calculating the number of fractional digits is the same as for the

corresponding simple operator `@`.

Rules: *R* = `scale`(*b*) and *R* = `scale`(*a*@*b*)

Example 1:

a is assigned a number with one fractional digit, *b* a number with two fractional digits: `scale`(*a*) is 1 and `scale`(*b*) is 2. If you inspect `scale`(*a*) after assigning the operands, `bc` returns a value of 2, which is `scale`(*b*). If you then inspect *a* again, the value returned is the value assigned to *b*:

```
$ bc
a=0.1
b=0.12
scale(a)
1
scale(b)
2
a=b
scale(a)
2
a
.12
```

Example 2

a is assigned a number with two fractional digits, *b* a number with three. In the function call, *a* is assigned the value resulting from adding the two operands. `bc` sees the number of fractional digits in the result of an addition as being equal to the number of fractional digits in the operand with the most fractional digits. After assignment *a* has the same number of digits, i.e. three:

```
$ bc
a=0.12
a=0.123
scale(a)
2
scale(b)
3
a=+b
scale(a)
3
a
.243
```

`%` With remaindering, if the value of scale is non-zero, the result is computed as follows:

```
a%b = a - (a / b) * b
```

First the division is performed using the precision of `scale`. The precision of the multiplication by *b* is equal to:

`scale` + *B*

In other words the multiplication is performed with full precision. Thus here the `%` operator can be used as a measure of the precision with which the division is performed.

Finally *R* is whichever is the higher of *A* and (`scale` + *B*).

Rule: *R* = *max*((`scale` + *B*), *A*)

Example:

`scale` has a value of 4, *a* and *b* each have one fractional digit. The result of remaindering has five fractional digits:

```
$ bc
scale=4
```

```
a=1.2
b=1.1
scale(a%b)
5
a%b
.00001
```

EXIT STATUS

`0` All input files were processed successfully

`≠0` An error occurred during execution of `bc`

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

Addition, subtraction, multiplication, and division of numbers. Non-integer results are to have 2 digits after the decimal point.

```
$ bc
scale=2
3+7
10
8-15
-7
7*6
42
3/5
.60
quit
$
```

Example 2

Defining a function to compute an approximate value of the exponential function.

```
scale=20
define e(x){
      auto a, b, c, i, s
      a = 1
      b = 1
      s = 1
      for(i=1;1==1;i++){
              a = a*x
              b = b*i
              c = a/b
              if(c==0) return (s)
              s = s+c
      }
}
```

The termination criterion of the for loop is contained in the body of the loop [`if(c==0)`] and is not given by the second expression of the for statement as is usually the case. In a C program, the second expression of the for statement would simply be omitted; but in a `bc` program the for statement must always have 3 expressions. The expression `1==1`, which is always true, has been inserted for this reason.

Example 3

Printing approximate values of the exponential function of the first ten integers.

```
for(i=1;i<=10;i++) e(i)
```

Example 4

Printing the squares of the integers 1 to 4:

```
$ bc
for(i=1;i<5;i++) {i*i}
1
4
9
16
quit
```

Example 5

Printing the input to `dc` for calculating the squares of the integers 1 to 4.

Input:

```
$ bc -c □
for(i=1;i<5;i++) {i*i}
```

Output:

```
[lili*ps.lid1+sis.li 5>0]s0 □
1dsis.li 5>0
```

FILES

`/usr/lib/lib.b`
: Math library

`/usr/bin/dc`
: Desk calculator

SEE ALSO

`dc`(1).

User's Reference Manual

bdiff(1)

NAME

`bdiff -` compare large files (big diff)

SYNOPSIS

`bdiff` *file1 file2* [*n*] [`–s`]

DESCRIPTION

`bdiff` compares two files in a manner analogous to `diff(1)` , but it can handle files that are too large for `diff`.

The output from `bdiff` (as with `diff`) consists of:

- the lines in which the specified files differ,
- the `ed` commands required to convert *file1* into *file2* [see `ed(1)`].

OPTIONS

The arguments and options must appear in the above order.

file1 file2

file1 and *file2* are the names of the files that you want to compare. If you use a dash (–) instead of a file name, `bdiff` reads from standard input.

n In *n* you can specify how many lines each segment is to have when `bdiff` divides the files.

n is an integer greater than 0.

n not specified: *n* = 3500

`–s` Some `bdiff` error messages are suppressed. This does not, however, suppress error messages issued by `diff`. This option is primarily used in shell scripts.

Caution:

If the `–s` option is specified in addition to a value for *n*, the correct order must be observed: *n* must come before `–s`.

Functionality

`bdiff` ignores lines common to the beginning of both files. Starting with the first line in which the files differ, `bdiff` divides the specified files into *n*-line segments. The default value for *n* is 3500, but you can change this value when you call `bdiff` (see the *n* option). `bdiff` invokes `diff`. `diff` then compares the corresponding segments. You should therefore choose a value for *n* which will not create segments too large for `diff` to handle.

`bdiff` can only find differences between segments with corresponding segment numbers: it does not, for example, compare segment 4 of *file1* with segment 5 of *file2*. `bdiff` thus often gives a more complicated picture of the differences than you might wish.

Despite the subdivision into segments, output line numbering is consecutive from the beginning to the end of the files. The files thus appear to be have been processed whole.

EXAMPLES

This example is intended to demonstrate the effect of dividing the files into segments. First a file `abc1` with 26 lines is created. Each line has a consecutive lowercase letter of the alphabet in column 1; thus, `a` is in line 1, `z` in line 26. This file is then copied to the file `abc2`, and a new line with the digit `0` in column one is inserted before the first line of `abc2`. The two files are then compared in 5-line segments.

```
$ bdiff abc1 abc2 5
0a1
> 0
```

```
5d5
< e
5a6
> e
10d10
< j
10a11
> j
15d15
< o
15a16
> o
20d20
< t
20a21
> t
25d25
< y
25a26
> y
```

To start with, the first 5 lines of the two files are compared. The following actions are required to make the first 5 lines of `abc1` and `abc2` match:

- The first line of `abc2` (digit `0`) must be inserted before the first line of `abc1`. `bdiff` prints the `ed` command that you need to do this: `0a1`.

  (This `ed` command actually inserts line 1 of `abc2` after line 0 of `abc1`).
- The fifth line of `abc1` contains the letter `e` and must be deleted, since `abc2` does not include any lines with the letter `e` in its first 5 lines: `5d5`.

Lines 6 – 10 of the two files are compared next. Line 6 of `abc2` (letter `e`) must now be inserted before line 6 of `abc1` (letter `f`), and line 10 of `abc1` must be deleted. The segments will then be identical.

Since `bdiff` compares files in segments, its output is generally more complicated than that of `diff`. In effect, inserting the first line of `abc2` before the first line of `abc1` would have been sufficient.

SEE ALSO

`cmp(1)`, `comm(1)`, `diff(1)`, `diff3(1)`, `ed(1)`.

User's Reference Manual

bfs(1)

NAME

`bfs` - big file scanner

SYNOPSIS

`bfs` [–] *file*

DESCRIPTION

`bfs` is similar to `ed`, except that you cannot use it to make changes to the file. `bfs` is read-only, but it can process much larger files than `ed`.

The maximum file size is 1024 Kbytes. The maximum number of lines is 32,000; the maximum number of characters per line is 512 (including the newline character).

`bfs` is usually more efficient than `ed`, since the file is not copied to a buffer.

`bfs` commands may be entered interactively or read from a file (see `xf` below).

– suppresses the usual displaying of
- the size of a file
- error messages

file name of the file that you want to scan with `bfs`.

Output

`bfs` prints the size of the file being scanned as well as the size of any file written with the `w` command.

As with `ed`, entering `P` <RETURN> causes an asterisk to be used as the input prompt. If errors occur in * mode, `bfs` issues an error message and exits. Entering `P` <RETURN> again deactivates the display of the * prompt and the error messages.

Addresses

All address formats described under `ed` are supported. The following additional options are available:

- Regular expressions can be enclosed in >...> and <...< in addition to /.../ and ?...?. In this context, > indicates a downward search to the end of the file, while < indicates an upward search to the beginning of the file. However, in contrast to / or ?, the search does not wrap around from the end of the file to the beginning or vice versa.
- Only the letters `a` to `z` may be used to mark lines. All 26 marks are remembered (stored).

Commands

There are some commands that function as in `ed` and others that are only available with `bfs`. The two types are given below in separate alphabetical listings.

ed-like commands

The following commands function as in `ed`, with the exception of a few minor differences which are indicated at the appropriate place. A detailed description of these commands is provided under `ed`.

`e` *file*
You must always specify a file. `bfs` treats this *file* as the current file; only one file can be processed at a time.

`f` Prints the name of the file being scanned. There is no remembered file name.

`p` The commands `1,10` and `1,10p` have the same meaning; the first 10 lines are printed.

`P` Error messages are given in response to errors.

`q`

`w` `w` is independent of output redirection, truncation, or crunching (see `xo`, `xt`, `xc` below).

Additional bfs commands

[*range*]xb/*regular_expression*/*label*

range not specified:

The search starts at the current line.

If a string matching *regular_expression* is found in the specified range, the line containing the string becomes the current line, and a jump (either downward or upward) is made to *label*. Labels are set with the colon command :*label* (see below) and are located in the bfs command file (see xf).

A jump is not possible in the following situations:

- either of the addresses in *range* is not between 1 and $ (the first and last lines of the file).
- the second address is less than the first.
- the regular expression does not match at least one line in the specified range (including the first and last lines).

xb is the only command which does not issue error messages in response to bad addresses. In such instances, it does nothing; and if it does nothing, you know that the specified address is invalid. It is therefore possible to use this command to test whether addresses are bad before other commands are executed.

The command

```
xb/^/label
```

initiates an unconditional jump, i.e. no condition needs to be fulfilled.

The xb command is allowed only if it is **not** read from a terminal. If it is read from a pipe, only a downward jump is possible.

xbn*label*

xbz*label*

These two commands test the last saved return code of a UNIX system command (!*command*):

- xbz jumps to the specified label if the return code is 0 (true).
- xbn jumps to the specified label if the return code is non-zero (false).

Otherwise, xbn and xbz have no effect.

xc[*switch*]

Activates/deactivates crunching of output from the p or null commands. Crunched output has strings of tabs and blanks reduced to one blank; blank lines are suppressed.

switch can have the values:

0 Output is not crunched (default).

1 Output is crunched.

switch not specified:

The setting of the switch is toggled.

xf*file*

bfs reads further commands from the named *file*. When the end-of-file character is reached, an interrupt signal is received or an error occurs, bfs reads the next command in the file containing the xf command.

xf commands may be nested to a depth of 10.

xn Lists the marks currently in use. Line marks are set by the k command (see ed).

xo[*file*]

Further output from the p and null commands is redirected to the named *file*. If the file does not exist, it is created and assigned the following permissions:

```
rw-------
```

Note: The permissions of a new file are controlled by umask.

If the file already exists, it is truncated to a length of 0 and overwritten (if necessary); its permissions are not altered. Write-protected files are not overwritten.

file not specified:

Output is redirected to standard output. If you wish to redirect output to two different files in succession, you will have to redirect it to standard output between the two operations.

`xt`*n* Output from the `p` and null commands is truncated to at most *n* characters.

`xt`*n* not specified:

Output is restricted to 255 characters.

`xv`*digit*^[ ][*value*]

`xv` lets you assign a value to a variable. The variable name is specified by the *digit* following the `xv` command. *digit* can be any number between 0 and 9.

The commands `xv5100` and `xv5 100` both assign the value `100` to the variable `5`. The command `xv61,100p` assigns the value `1,100p` to the variable `6`.

To reference a variable, put a `%` in front of the variable name. Thus using the assignments noted above for variables `5` and `6`, each of the following commands:

```
1,%5p
1,%5
%6
```

will print the first 100 lines.

Entering the command:

```
g/%5/p
```

initiates a global search for the string `100` and prints each line containing these characters. The special meaning of the percent sign can be escaped by preceding it with a backslash, i.e. by entering `\%`.

The command:

```
g/".*\%[cds]/p
```

matches and lists all strings that contain double quotes, followed by any number of characters, followed by a percent sign preceding the letter `c`, `d` or `s`. You can thus use this command to search for `printf` statements such as are used in `awk` or C.

`xv` also makes it possible for you to store the first line of output from a UNIX system command into a variable. For this purpose, the first character of *value* must be an exclamation mark.

The value of variable `6` could thus be incremented by one with the aid of the `expr` command:

```
xv6!expr %6 + 1
```

A somewhat more complicated example:

```
.w tom
xv5!cat tom
!rm tom
!echo "%5"
```

The current line is written to the file `tom` from where it is then read with `cat` and stored in variable `5`. The file `tom` is then deleted. The stored line is displayed by the `echo` command [see specialchar(5)].

The special meaning of ! as the first character of *value* is escaped if it is preceded by a backslash. Thus,

```
xv7\!date
```

stores the value `!date` in variable 7.

value not specified:

The variable is assigned the number of the current line.

`:`*label*

A*label* is a string that serves as a reference point for the `xb`, `xbn`, and `xbz` commands. You can write a *label* in a command file (see `xf`).

label is terminated by a newline character. Blanks between the colon and the start of *label* are ignored. The `:` command may also be used to insert comments into a command file, since labels need not be referenced.

ERROR MESSAGES

A question mark ? is output for errors in commands if `bfs` is not in * prompting mode (see the `ed`-like commands section, `P` command).

Self-explanatory error messages are output if `bfs` is in * prompting mode.

EXAMPLES

How labels work

Create a file named `animals` with the five lines given below (line numbers are included in the file for the sake of clarity):

```
1 Dogs
2 Cats
3 Mice
4 Horses
5 Cows
```

You will also need to create a `bfs` command file named `cmd1` so that commands can be read from it later. You first want to search for a line that has the word `Mice` in it. If `Mice` is not found, all lines are to be printed; if `Mice` is found, only the line containing `Mice` is to be printed. Then `bfs` is to terminate. Thus `cmd1` must contain the following:

```
1,$xb/Mice/found
1,$p
xb/^/continue
:found
.p
:continue
q
```

If `Mice` is not found, the instruction to print the current line must be skipped, since the last line would otherwise be printed twice (by the command `1,$p` in line 2 and by the command `.p` in line 5). The `xb/^/continue` command therefore effects an unconditional jump to skip line 5.

After both files have been created, they are used as follows:

```
$ bfs animals
37
xfcmd1
3 Mice
$
```

If you want, you can now change `Mice` to `Lice` in either `cmd1` or `animals`. All lines in `animals` will then be printed.

How xbz and xbn work

First a file is to be searched for five occurrences of the string `Mice`. This entails creating the `bfs` command file `cmd2` with the following contents:

```
xv55
:1
/Mice/
xv5!expr %5 - 1
![ %5 != 0 ]
xbz 1
q
```

The digit variable `5` is assigned a value of `5`. The string `Mice` is then matched and printed once. Next the value in the digit variable `5` is decremented by 1. A check is then made to determine whether the value of `5` is still non-zero. If it is, a jump is made to label `1`; if the value of `5` is 0, `bfs` is terminated.

The file `animals` will again be used for the search:

```
$ bfs animals
37
xfcmd2
3 Mice
3 Mice
3 Mice
3 Mice
3 Mice
$
```

The string `Mice` is printed five times, since /.../ continues searching beyond the end of a file by starting once again at the beginning.

If `xbn` is used, `cmd2` must have the following contents in order to produce the same results:

```
xv55
:1
/Mice/
xv5!expr %5 - 1
![ %5 = 0 ]
xbn 1
q
```

SEE ALSO

`ed`(1).

User's Reference Manual

biff(1)
BSD Compatibility Package

NAME

`biff -` give notice of incoming mail messages

SYNOPSIS

`/usr/ucb/biff [y|n]`

DESCRIPTION

`biff` turns mail notification on or off for the terminal session. With no arguments, `biff` displays the current notification status for the terminal.

The `y` option allows mail notification for the terminal. The `n` option disables notification for the terminal.

If notification is allowed, the terminal rings the bell and displays the header and the first few lines of each arriving mail message. `biff` operates asynchronously. For synchronized notices, use the `MAIL` variable of `sh(1)` or the `mail` variable of `csh(1)`.

A `biff y` command can be included in your `~/.login` or `~/.profile` file for execution when you log in.

FILES

`~/.login`
`~/.profile`

SEE ALSO

`csh(1)`, `mail(1)`, `sh(1)`.

cal(1) - cut(1)

User's Reference Manual

cal(1)

NAME

`cal -` print calendar

SYNOPSIS

`cal` [[*month*] *year*]

DESCRIPTION

`cal` writes a calendar to standard output.

OPERANDS

No argument specified:
`cal` prints the calendar for the current month.

month
`cal` prints the calendar for the specified month.
The possible values for *month* are 1 to 12.
If you specify a value for *month*, you must also specify a value for *year*.
month not specified:
`cal` prints the calendar for the entire year.

year The calendar for the specified year is printed.

The possible values for *year* are 1 to 9999.

Please note that `cal 88`, for example, refers to 88 A.D., not 1988.

Note:

If you call `cal` using the argument `1752` or the arguments `9 1752`, the output for September is shorter than usual. This is because `cal` allows for an adjustment of 11 days which took place in that month in 1752.

EXIT STATUS

The exit status is 0 when successful and > 0 when the values specified for *year* or *month* lie outside the permissible range.

ERROR MESSAGE

`Bad argument [ for month | for year ]`

The values you have specified for *year* or *month* are not within the permissible range.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_CTYPE` governs character classes, character conversion (shifting) and the behavior of character classes in regular expressions.

If `LC_MESSAGES` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Print the calendar for February 1996 in the C locale:

```
$ cal 2 1996
   February 1996
Su Mo Tu We Th Fr Sa
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29
```

User's Reference Manual

calendar(1)

NAME

`calendar -` reminder service

SYNOPSIS

`calendar [-]`

DESCRIPTION

The `calendar` command is a convenient way of reminding yourself of important dates or appointments. You need to set up a file called `calendar` containing all the necessary details. Then when you call the `calendar` command, all lines in the `calendar` file that contain today's or tomorrow's date anywhere in the line are printed on standard output.

The `calendar` command has to be called from the directory that contains the `calendar` file. If you would like to be reminded of your appointments automatically whenever you log in, set up your `calendar` file in your home directory and put the `calendar` command in your `.profile` file. Then today's and tomorrow's appointments will be printed on standard output whenever you log in.

`calendar` can also be invoked regularly by using the `at` and `crontab` commands.

Dates in the following formats are recognized: `Aug. 31`, `august 31` and `8/31`. On Fridays, `tomorrow` extends through Saturday, Sunday and Monday. This extended idea of tomorrow does **not** account for holidays. In other words, if today is the day before a holiday, `tomorrow` is interpreted as usual and includes only the holiday.

OPTIONS

`-` If the `calendar` file is in your home directory, you can use the `-` option to call the `calendar` command from any of your directories.

Your appointments are then not written to standard output, but are sent by `mail` instead.

Only for the system administrator:

If the system administrator calls `calendar` with the `-` option, all users who have a `calendar` file in their home directory will be sent their appointments by `mail`. This is normally done on a daily basis by the Reliant UNIX operating system, which informs all users of appointments noted in the `calendar` file.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_TIME` environment variable governs the format of date and time strings.

If `LC_MESSAGES` or `LC_TIME` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables were set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Mr. Calhoun writes the following appointments in his `/home/calhoun/calendar` file:

```
10/09 dentist 1 pm
10/10 pick up Mr. Swift from the airport at 5.30 pm
10/25 Uncle Albert's birthday
```

He writes the following command call in `/home/calhoun/.profile`:

```
calendar
```

When he logs in on October 8, the following text appears on the screen:

```
10/09 dentist 1 pm
```

When he logs in on October 9, the following text appears on the screen:

```
10/09 dentist 1 pm
10/10 pick up Mr. Swift from the airport at 5.30 pm
```

When he logs in on October 10, the following text appears on the screen:

```
10/10 pick up Mr. Swift from the airport at 5.30 pm
```

When he logs in on October 24 and 25, the following text appears on the screen:

```
10/25 Uncle Albert's birthday
```

FILES

`/usr/lib/calprog`
: Routine to work out today's and tomorrow's date

`/etc/passwd`
: File containing all installed login names

`/tmp/cal*`
: Temporary file

SEE ALSO

at(1), crontab(1), date(1), mail(1), cron(1M), environ(5).

User's Reference Manual

cat(1)

NAME

`cat -` concatenate and print files

SYNOPSIS

`cat [ –s ] [ –u ] [ –v[et ]] [ –– ] [ *file* ... ]`

DESCRIPTION

The `cat` command reads files in sequence and writes them to standard output. `cat` has no effect on the sequence and format of the characters in the files.

If you name more than one file when calling `cat`, these files are output sequentially in the specified order.

If you do not name a file, `cat` reads from standard input.

OPTIONS

No option specified:

Output is buffered in `BUFSIZ`-byte blocks. The value of `BUFSIZ` is governed by the machine you are working on. It is defined in the `/usr/include/stdio.h` file and may be 512, 1024, 4096 or 8192 bytes. If the files named on the command line do not exist, `cat` tells you that it cannot open them.

`–s` Messages reporting that files do not exist are suppressed.

`–u` Output without buffering, one byte at a time.

`–v` Non-printing characters (with the exception of tabs, newlines and form feeds) are written to standard output in the following form:

- Control characters: `^`*x* (CTRL-*x*)

  The possible values for *x* in the ISO 646 character set are shown in the table in `ascii(5)` .
- DEL (octal 177): `^?`
- Non-ASCII characters (with the high bit set): `M–`*x*, where *x* is the character defined by the 7 remaining bits.

`–e` Only works together with the `–v` option.

A dollar character `$` appears at the end of each line (prior to the newline character).

`–t` Only works together with the `–v` option.

Tabs appear as: `^I`, form feeds appear as: `^L`

`––` If *file* begins with a dash (–), the end of the command-line options must be marked with ––.

file Name of the file that is to be printed. You may specify more than one file. If you use a dash as the name for *file*, `cat` reads from standard input.

file not specified:

`cat` reads from standard input.

Caution:

Redirecting the output of `cat` to one of the files being read will result in the loss of that file's original contents. In the following command, for example, the contents of `file1` are lost:

```
cat file1 file2 file3 > file1
```

ERROR MESSAGES

`cat >`*outfile* □
outfile`: cannot create`

You do not have write permission for the output file *outfile* or for the directory containing *outfile*.

`cat` *infile* □
`cat: cannot open` *infile*

You do not have read permission for the input file *infile*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_CTYPE` governs character classes, character conversion (shifting) and the behavior of character classes in regular expressions.

If `LC_MESSAGES` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Concatenate and redirect the output of two files:

```
$ echo Monday Tuesday Wednesday >file1
$ echo Thursday Friday Saturday >file2
$ cat file1 file2 > file3
$ cat file3
Monday Tuesday Wednesday
Thursday Friday Saturday
```

Example 2

Display the contents of `file1`

```
$ cat file1
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
```

Now write two lines of text into `file2`

```
$ cat > file2
Through caverns measureless to man
Down to a sunless sea.
```
<CTRL-D>

Now move the contents of `file1` and `file2` to `file3`, add two lines from standard input, and then print the contents of `file3`.

```
$ cat file1 file2 - > file3
For he on honey dew hath fed
And drunk the milk of paradise.
```
<CTRL-D>
```
$ cat file3
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
For he on honey dew hath fed
And drunk the milk of paradise.
```

SEE ALSO

`cp(1)`, `pg(1)`, `pr(1)`, `ascii(5)`.

User's Reference Manual

cd(1)

NAME

`cd -` change working directory

SYNOPSIS

`cd` [*directory*]

DESCRIPTION

The shell built-in `cd` makes the specified directory your current working directory.

The `cd` command is rejected in a restricted shell [see `sh(1)`].

The Korn shell `ksh` has additional command-line options [see `ksh(1)`].

OPERANDS

directory

Name of the directory that is to become your current working directory. You must have execute permission for this directory. If you specify a relative or absolute path name for *directory*, you must have execute permission for all the directories that make up this path name.

If the name of the specified directory begins with one of the following characters, the command looks for the directory without reference to the `CDPATH` environment variable [see `sh(1)`]:

`/` means that the search begins in the root directory.

`./` means that the search begins in the current directory.

`../` means that the search begins in the parent directory.

If the name of the specified directory does not begin with any of the above characters, `cd` evaluates the `CDPATH` environment variable:

- If the `CDPATH` variable has not been defined or is null, `cd` looks for the specified directory relative to the current working directory.
- If the `CDPATH` variable has been assigned a value, `cd` looks for the specified directory sequentially in the directories whose paths are defined in the `CDPATH` variable. On finding the directory, `cd` writes the absolute path name of this directory on standard output before switching to it.

directory not specified:

The `cd` command puts you in your home directory. The home directory is identical to the login directory unless there is a different path name assigned to the shell variable `HOME`.

ERROR MESSAGES

file`: does not exist`

The specified directory does not exist. You can verify this with `ls -l`.

file`: not a directory`

Your argument is not a directory. This can also be verified with `ls -l`.

file`: permission denied`

You do not have execute permission for the specified directory.

If you have specified a relative or absolute path name for *directory*, you do not have execute permission for one of the directories which make up this path name.

`cd: restricted`

`cd` has been rejected because you are working in a restricted shell.

ENVIRONMENT VARIABLES

`HOME` contains the absolute path name of your home directory.

`CDPATH`

You can assign to `CDPATH` the absolute path names of directories that `cd` is to search.

By default this variable is undefined.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

Example 1

The following entry makes the subdirectory `dates` the current directory:

```
$ cd dates
$ pwd
/home/hofu/dates
```

Example 2

User `hofu` has redefined the `CDPATH` environment variable. He now wishes to change to his subdirectory `usr`, but with the following commands ends up in the directory `/usr` instead:

```
$ echo $CDPATH
/:/home/hofu/dates:.
$ pwd
/home/hofu
$ ls –l
drwx--x--x 2 hofu        144 Feb 28 12:32 usr
drwx--x--x 2 hofu        192 Feb 28 11:51 dates
-rw------- 1 hofu      11734 Mar  7 16:22 tests
. . .
$ cd usr
$ pwd
/usr
```

The `usr` directory is first looked for in the directories whose path names are assigned to the `CDPATH` variable. In this case, `CDPATH` contains a / for the root directory as the first path name. The current directory is the last to be searched by `cd`.

User `hofu` can prevent `cd` from evaluating the `CDPATH` environment variable by formulating the command in the following way:

```
$ cd ./usr
$ pwd
/home/hofu/usr
```

NOTES

Some differences in behavior may occur when using `cd`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh(1)`, `pwd(1)`, `sh(1)`, `chdir(2)`.

User's Reference Manual

ced(1)

NAME

`ced` - full-screen text editor

SYNOPSIS

`ced` [*option*] [*file*]

DESCRIPTION

In a full-screen text editor, unlike a line editor, you can use the arrow keys <CURSOR-UP>, <CURSOR-DOWN>, <CURSOR-RIGHT>, <CURSOR-LEFT>, <HOME> and <ENTER> to move the cursor anywhere on the screen and can then enter or modify text at that position. `ced` is menu-driven and provides you with various editing modes. Depending on the mode you select you can:

- enter text directly or
- enter commands for various editing functions, e.g. copy, delete etc.

Layout of this description

The description of `ced` is subdivided into seven sections:

- Command overview
- Working with `ced`
  - The screen
  - The window
  - The control panel
  - Moving the cursor
  - Selecting the mode
- Description of the modes
- Error messages
- Files
- Locale
- Environment variables

The standard version of `ced`, in which the menu functions and error messages appear in English (see **LOCALE**), is described below.

OPTIONS

`-f` *line-number*

`ced` positions the cursor on the line indicated by *line-number*.

`-s` *search-string*

`ced` positions the cursor on the first line containing *search-string*. If *search-string* is not present in the file, `ced` displays the start of the file and positions the cursor on the first line.

file Name of the file you want to edit. If there is no file with this name in the working directory, `ced` creates a new file.

file not specified:

`ced` opens the file that you were last editing with `ced` on your terminal and positions the cursor at the position it last occupied. It retrieves this information from the file `/var/ced/CE`*ttyname*`.`*loginname*.

If you have never opened a document with `ced` on your terminal before, `ced` will issue an error message and exit.

Note:

If the size of *file* grows beyond the user file size limit in the course of a `ced` session, *file* will no longer be backed up.

COMMAND OVERVIEW

The most important function keys, menu functions, and `ced` commands are summarized below in tabular form under different function groups. You will find more detailed descriptions in the section on **WORKING WITH CED**.

Loading `ced`

ced <RETURN>	Load `ced` with the file last edited
ced *file* <RETURN>	Load `ced` with the specified file

Quitting `ced`

<MENU> `qy`	Quit `ced` and save changes in current file
<MENU> `qn`	Quit `ced` without saving changes
<CTRL-@>	Quit `ced` (emergency exit) without saving changes

General editing functions

<MENU> `e`	Enter new text (overwrite mode)
<MENU> `i`	Insert new text into old (insert mode)
<MENU> `a`	Save text
<F9>	Save in current file
<MENU> `a` <RETURN>	Save in current file In control panel: *file*
<MENU> `a` <RETURN>	Save in specified *file*
<MENU> `qy`	Quit `ced` and save in current file
<MENU> `d`	Switch documents
<MENU> `d` <RETURN>	Switch to last file In control panel: *file*
<MENU> `d` <RETURN>	Switch to specified *file*
<F10>	Switch documents
<MENU> `c`	Execute commands In control panel: [*area*] *command* Values for *area*: *n*`l` lines from cursor position *n* paragraphs from cursor position – no input area; nothing deleted Default: cursor position to end of paragraph.
<MENU> `!`	Call shell (temporary escape to shell)
<MENU> `p`	Program keys
<MENU> `x`	Show programmed keys

<MENU> `t`	Find tag

Special text-editing functions

<MENU> `l`	Edit line
<MENU> `lb`	Insert a blank line
<MENU> `ld`	Delete and buffer a line
<MENU> `li`	Retrieve and insert the buffered line
<MENU> `lj`	Join two lines
<MENU> `lr`	Delete the remainder of a line
<MENU> `ls`	Split a line
<MENU> `b`	Edit line block
<MENU> `bb`	Insert blank lines
<MENU> `bc`	Copy the marked block
<MENU> `bd`	Delete and buffer the marked block
<MENU> `bi`	Retrieve and insert the marked block
<MENU> `bm`	Mark a line
<MENU> `r`	Edit rectangle
<MENU> `rd`	Delete a rectangle
<MENU> `rh`	Horizontal insertion
<MENU> `rm`	Mark a corner
<MENU> `ro`	Copy with overwrite
<MENU> `rv`	Vertical insertion

Positioning

<START>	Start of the file
<HOME><HOME><CURSOR-UP>	First line of the file
<HOME><HOME><CURSOR-DOWN>	Last line of the file
<PAGE-UP>	Up 9 lines (½ window)
<PAGE-DOWN>	Down 9 lines (½ window)
<F15>	Down 19 lines (1 window)
<F16>	Up 19 lines (1 window)
<MENU> `w`	Position window to a particular line or column; in control panel: n (number)
<RETURN><CURSOR-UP><CURSOR-DOWN>	line number n
<CURSOR-LEFT><CURSOR-RIGHT>	column number n relative to the current position; in control panel: $+n$ or $-n$
<RETURN><CURSOR-DOWN>	n lines down
<CURSOR-UP>	n lines up

<CURSOR-DOWN>	*n* lines down
<CURSOR-LEFT>	*n* columns to the left
<CURSOR-RIGHT>	*n* columns to the right
<MENU> `s`	Search for string; in control panel: *search-string*
<CURSOR-DOWN>,<RETURN> or <F13>	Search forward
<CURSOR-UP> or <F14>	Search backward

Cursor positioning

<CURSOR-UP>	One line up
<HOME> <CURSOR-UP>	First line of window
<HOME> <HOME> <CURSOR-UP>	First line of file
<CURSOR-DOWN>	One line down
<HOME> <CURSOR-DOWN>	Last line of window
<HOME> <HOME> <CURSOR-DOWN>	Last line of file
<CURSOR-LEFT>	One character to the left
<WORD-RIGHT>	Tab left (tab stops are in columns 1, 9, 17 etc.)
<HOME> <CURSOR-LEFT>	Left-hand edge of window
<HOME> <HOME> <CURSOR-LEFT>	Start of line
<HOME> <WORD-RIGHT>	Start of line
<CURSOR-RIGHT>	One character to the right
<WORD-LEFT>	Tab right (tab stops are in columns 1, 9, 17 etc.)
<HOME> <CURSOR-RIGHT>	Right-hand edge of window
<HOME> <HOME> <CURSOR-RIGHT>	End of line
<HOME> <WORD-LEFT>	End of line

Keys

<INSERT CHAR.>	Insert characters
<DELETE CHAR.>	Delete characters
<INSERT LINE>	Insert lines
<DELETE LINE>	Delete lines
<BACKSPACE>	Backspace; delete character left of cursor
<CTRL-R>	Refresh screen
<CTRL-@>	Quit `ced` (emergency exit; files not saved)

Delete characters in control panel

<BACKSPACE>	Backspace; delete character left of cursor
<CTRL-X>	Delete everything entered

Standard functions

<START>	Start of file
<F9>	Save file
<F10>	Switch documents
<F11>	Mark line (2 marks)
<F12>	Copy line
<F13>	Search forward
<F14>	Search backward
<F15>	Page down
<F16>	Page up
<F17>	Retrieve deleted line or line block

WORKING WITH CED

`ced` works on a copy of the file you want to edit. `ced` does not transfer the changes to the file itself until you

- quit `ced` and enter `y` to save your work (see `q` mode) or
- save the changes in `a` mode. In this case you can also choose a different file name.

The screen

```
** CED Text Editor V1.3   Line:   1  Column: 1  Name: file
@
@
@
@
@
@
@
@
@
@
@
@
You wish to enter text?
Please enter your text
```

The header line shows:

- the `ced` version string,
- the current line and column position of the cursor,
- the name of the file currently displayed in the window.

The window

The window shows an extract from the file you are editing (19 lines). Blank lines at the bottom of a file are indicated by a tilde (~) at the start of each line. The position of the cursor given in the header line tells you which part of the file you are currently editing.

You can shift the position of the window as shown below:

<PAGE-UP>	up 9 lines (½ window),
<PAGE-DOWN>	down 9 lines (½ window),
<F15>	down 19 lines (1 window),
<F16>	up 19 lines (1 window).

In `w` mode (position the window), you can shift the window by any amount up, down, left or right.

The window moves automatically when the cursor reaches the edge of the area displayed.

Lines that extend beyond the right edge have a continuation character > in the last column.

The control panel

In the control panel, `ced` shows you the mode you have selected and the functions that are available to you in this mode.

When you call `ced`, it always enters `e` mode (enter new text).

In some modes you can make entries in the control panel, e.g. the file name in `d` mode (switch documents) or the character string to be searched for in `s` mode (search for string).

An entry in the control panel can be corrected as follows:

<BACKSPACE>	deletes characters one at a time
<CTRL-X>	deletes everything you have entered

Moving the cursor

<CURSOR-UP>	up
<CURSOR-DOWN>	down
<CURSOR-RIGHT>	right
<CURSOR-LEFT>	left
<WORD-LEFT>	tab right; tab stops are in columns 1, 9, 17, 25 etc.
<WORD-RIGHT>	tab left
<HOME>	This key is used to move to the edges of the screen or document
<HOME> <CURSOR-RIGHT>	right-hand edge of the window
<HOME> <CURSOR-LEFT>	left-hand edge of the window
<HOME> <CURSOR-UP>	first line of the window
<HOME> <CURSOR-DOWN>	last line of the window (unless end of file comes first)
<HOME> <WORD-LEFT>	first column of the screen shifted 80 columns to the right
<HOME> <WORD-RIGHT>	first column of the current window
<HOME> <HOME> <CURSOR-RIGHT>	last column (512)
<HOME> <HOME> <CURSOR-LEFT>	first column (1)
<HOME> <HOME> <CURSOR-UP>	first line of the file
<HOME> <HOME> <CURSOR-DOWN>	last line of the file

Selecting the mode

Press the <MENU> key followed by a letter key in order to select one of the functions listed below:

Key Function

<MENU> `a`
: save text

<MENU> `b`
: edit line block

<MENU> `c`
: execute command

<MENU> `d`
: switch documents

<MENU> `e`
: enter new text

<MENU> `i`
: insert text

<MENU> `l`
: edit line

<MENU> `p`
: program keys

<MENU> `q`
: quit `ced`

<MENU> `r`
: edit rectangle

<MENU> `s`
: search for string

<MENU> `t`
: find tag

<MENU> `w`
: position window

<MENU> `x`
: show programmed keys

<MENU>`!`
: call shell

<MENU> <HELP>
: displays a brief description of `ced` in English

You can page forward or backward in the brief description, just as in any other file. If you specify a character string in `s` mode (search for string), you can use key <F13> or <F14> to search for this string in the brief description.

`ced` normally processes inputs immediately without your having to press <RETURN>. Inputs in the control panel are the exception to this rule. The keys have different functions depending on the mode you are currently using.

Undoing a mode selection

If you have selected the wrong mode, press <MENU> again. This returns you to the initial menu from which you can reselect the mode you require.

DESCRIPTION OF THE MODES

<MENU> a – Save text

The following text appears in the control panel:

```
You wish to save your text?
Please enter a file name or press <RETURN>
```

In the control panel, enter the name of the file in which the text that you are currently editing is to be stored. Then press <RETURN>. If you do not enter a file name, `ced` writes the text into the current file. The name of the current file is shown in the header line. The original contents of the file are overwritten.

If you press <CTRL-D>, the following text appears in the control panel:

```
You wish to quit CED?
Would you like to save your text?: Enter y=yes or n=no
```

You can now continue as described under mode `q`.

<MENU> b – Edit line block

The following text appears in the control panel:

```
You wish to edit a line block?
Please select one of the following functions: (b,c,d,i,m, <MENU>)
```

In this mode you can edit blocks of consecutive lines. This entails marking the top and bottom lines of the range (function `m`).

The following functions are available:

`m` Mark a line:

Move the cursor to the required line and press the `m` key. `ced` issues the following message in the control panel: `MARK SET`. It is always the last two marks set that are valid. If a block of lines is deleted (function `d`), the marks are also deleted.

If you only wish to edit a single line, you can mark it by pressing the `m` key twice.

`b` Insert blank lines:

A number of blank lines equal to the number contained in the marked area are inserted above the cursor line.

`c` Copy the marked area:

A copy of the marked area is inserted above the cursor line.

`d` Delete and buffer the marked area:

The marked area is deleted from the file and buffered. Each subsequent delete and buffer operation overwrites the block last buffered.

`i` Retrieve and insert the buffered area:

The last block deleted and buffered (function `m`) is inserted above the cursor line.

If you press <CTRL-D>, the following text appears in the control panel:

```
You wish to quit CED?
Would you like to save your text?: Enter y=yes or n=no
```

You can now continue as described under mode `q`.

<MENU> c – Execute command

The following text appears in the control panel:

```
You wish to execute a command? -->
Please enter a command
```

In this mode you can call a Reliant UNIX command and pass an area of your file to it as input. `ced` replaces this area with the output from the executed command. You define the input area and the command to be executed in the input line of the control panel as follows:

[*area*] *command*

area Input area for the command. This area of the file is passed to the standard input of the command and deleted in the file.

area can be: *n*l, *n*, –, *

*n*l *n* lines from the cursor position.

n *n* paragraphs from the paragraph in which the cursor is located. A paragraph is an area between two blank lines or between the start of the file and the first blank line.

– Empty input area. Nothing is deleted from the file, and no data is passed to the standard input of the Reliant UNIX command.

* All the lines from the line containing the cursor to the end of the document

Caution:

If you put a blank in front of the asterisk (*), * will be executed as a Reliant UNIX command. As the shell expands the asterisk, there may be unexpected results.

area not specified:

The input area is the area from the cursor to the end of the paragraph.

command

The command to be executed. If there is an invalid argument, `ced` issues a message and does not delete the input area. All Reliant UNIX commands are allowed, with the exception of commands such as `login`, `su`, and any other commands that change the shell environment.

The output from the command is inserted at the point from which the input has been deleted. If you do not want anything to be deleted, you should specify an empty input area (0l or 0 or simply –).

Reliant UNIX commands that do not expect input (e.g. cat *file*) must not be passed an input area as this could result in malfunctioning. In this case you should always specify an empty area. If the `SHELL` variable is not set, Reliant UNIX commands cannot be executed.

<MENU> d – Switch documents

The following text appears in the control panel:

```
You wish to switch documents? -->
Please enter a file name or press <RETURN>
```

In the control panel, enter the name of the other file you want to edit. Then press <RETURN>. In the window you see the start of the new file. `ced` is now in `e` mode (enter new text).

If the file you name does not exist, `ced` asks whether the file is to be created. If you respond with y, a new file is created.

Editing files in parallel

If you have opened several files as described above, you can:

- toggle from one file to another:

  Just press <RETURN> in `d` mode. `ced` to the file you worked on last.
- transfer text from one file to another:
  - Areas you have marked (in mode `b`, `r` or `l`) in one file can be accessed directly from the other if you switch to the same mode again.
  - Buffered blocks can be retrieved in the other file (mode `b` or `l`).

If you switch to `q` mode (quit `ced`), `ced` asks you if you want to have the current file saved. `ced` then does the same for the next open file in which you have made changes, and so on. In this way you can close all files edited during a `ced` session.

<MENU> e – Enter new text

The following text appears in the control panel:

```
You wish to enter text?
Please enter your text
```

This is text entry mode. In this mode, you may overwrite any text that is already in the file.

The following keys (in the INSERT or DELETE block as appropriate) may prove useful:

<CHAR.> Insert or delete individual characters

<LINE> Insert or delete individual lines

If you press <CTRL-D>, the following text appears in the control panel:

```
You wish to quit CED?
Would you like to save your text?: Enter y=yes or n=no
```

You can now continue as described under mode `q`.

<MENU> i – Insert text

The following text appears in the control panel:

```
You wish to insert something in the text?
Please insert your text
```

In this mode you can insert text. The existing text located at the cursor position is shifted to the right, beyond the edge of the window if necessary. The maximum possible length for a line is 512 characters. `ced` issues a message if the line gets too long.

If you press <CTRL-D>, the following text appears in the control panel:

```
You wish to quit CED?
Would you like to save your text?: Enter y=yes or n=no
```

You can now continue as described under mode `q`.

<MENU> l – Edit line

The following text appears in the control panel:

```
You wish to edit a line?
Please select one of the following functions: (b,d,i,j,r,s, <MENU>)
```

Use the cursor to select the line you want to manipulate.

`b` Insert a blank line:

A blank line is inserted above the cursor line.

`d` Delete and buffer the line:

The selected line is deleted and buffered. Each subsequent delete and buffer operation overwrites the line last buffered.

`i` Retrieve and insert the buffered line:

The last line buffered is inserted above the cursor line.

`j` Join two lines:

The selected line and the line that follows it are joined to form one line.

`r` Remove the remainder of the line:

The remainder of the line from the cursor onward is deleted.

`s` Split the line:

The rest of the line from the cursor position onward is written on a new line.

If you press <CTRL-D>, the following text appears in the control panel:

```
You wish to quit CED?
Would you like to save your text?: Enter y=yes or n=no
```

You can now continue as described under mode `q`.

<MENU> p – Program keys

The text of the menu used last is displayed in the control panel.

In this mode you can assign character strings, text modules, cursor movements, and mode changes to certain keys.

Although you can program almost any key in this way, preference should be given to the function keys. When you subsequently press a key that you have programmed, the character strings or instructions corresponding

to it are passed to `ced`.

Recording macros

When you switch to "Program keys" mode, the word PROTO appears in the header line. Everything that you do from here on is stored: text that you enter, mode changes, etc.

To assign these actions to a key, press the <MENU> and <p> keys again. `ced` displays the message:

```
****PRESS KEY TO BE CHANGED
```

Now press the key to which you wish to assign the stored actions. `ced` switches back to `e` mode (enter new text).

Restricted keys

You can program all keys except <MENU> and <p>.

Preference should be given to keys <F1>, <F2> etc.

Note that although keys <F9> to <F17> are bound to standard functions by `ced` (see **Keys with special functions** below), they can be reprogrammed for other purposes. Reprogrammed keys lose their original functions until restored by you, and as long as you save your text at the end of your session they stay reprogrammed for future `ced` sessions.

When you quit `ced`, your key assignments are automatically stored for your login name (in the file `/var/ced/CE`*key*`.`*loginname*). The next time you invoke `ced`, the key assignments you programmed in your last `ced` session will come into effect again.

Cancelling key assignments

Since all keys have their original function assignments in mode `p`, you can undo a macro assignment and restore the original function of the key as follows:

Press <MENU> <p> (*Program keys* mode) followed by the key whose assignment you wish to undo. The original function of the key is now stored for the pending assignment. You then press <MENU> <p> and the key to be programmed, i.e. the key whose assignment is to be restored.

<MENU> q – Quit ced

The following text appears in the control panel:

```
You wish to quit CED?
Would you like to save your text?: Enter y=yes or n=no
```

In `q` mode you can choose whether or not to save the changes that you have made. Type `y` or `n`:

`y` (yes) You exit the editor and the changed file is saved; all changes are now incorporated into your file.

`n` (no) You exit the editor without saving the changes, i.e. the data that was originally in your file remains unaltered. Empty files are deleted at the end of the session unless they were already empty before the session began.

If after pressing <MENU> `q` you do not wish to terminate the `ced` session but wish to continue, simply press <MENU> again. You will then be shown the selection menu and you may continue normally.

<MENU> r – Edit rectangle

The following text appears in the control panel:

```
You wish to edit a rectangle?
Please select one of the following functions: (d,h,m,o,v, <MENU>)
```

A rectangle is an area of a file that you define by marking two diagonally opposite corners (function `m`).

`m` Mark a corner:

The position at which the cursor is located is marked when you press the `m` key. `ced` displays the message: `MARK SET`. The valid marks are always the last two marks set. Deleting a rectangle (`d` function) also deletes the marks.

`d` Delete a rectangle:

The marked rectangle is deleted. It is **not** buffered anywhere. If you wish to reposition a rectangle, copy

it first (function `h`, `o` or `v`) and then delete the original.

`h` Horizontal insertion:

The marked rectangle is inserted, with its top left-hand corner at the cursor position. Text in the lines involved from the cursor column onward is shifted to the right.

`o` Copy with overwrite:

The marked rectangle is copied with its top left-hand corner at the cursor position, overwriting existing text.

`v` Vertical insertion:

The marked rectangle is inserted above the line marked by the cursor. The left-hand edge of the rectangle is on the column in which the cursor is located.

If you press <CTRL-D>, the following text appears in the control panel: `You wish to quit CED?`

```
Would you like to save your text?: Enter y=yes or n=no
```

You can now continue as described under mode `q`.

<MENU> s – Search for string

The following text appears in the control panel:

```
You want to search for something? -->
Please enter the search string
```

You enter the character string that you wish to search for on the control panel. The search can be started with one of the following keys:

<CURSOR-DOWN> or <RETURN>	Search forward towards end of file
<CURSOR-UP>	Search back towards start of file

If the character string does not occur again before end/start of file as appropriate, `ced` displays the message: `TEXT NOT FOUND`.

Leading and trailing spaces are significant. Unlike some other editors, `ced` does not recognize regular expressions.

The search function can be terminated with the <DEL> key at any time.

<MENU> t – Find tag

If you have several C, Pascal, FORTRAN, lex or yacc source files that are part of a large program suite, you can use the `ctags` command to create a tag file containing the names and precise locations (i.e. file and position) of all functions belonging to this program suite.

In "Find tag" mode, you enter the name of the function you want to go to in the control panel. `ced` switches to the appropriate file and positions the cursor on the definition of that function. The tag file must be in the current directory and contain the name of the specified function.

<MENU> w – Position window

The following text appears in the control panel:

```
You wish to position the window? -->
Please enter line, column or relative position (with prefix)
```

You can shift the window in two ways in this mode:

- to a particular line or column:

  Enter an unsigned number in the control panel. The number must not be larger than the total number of lines in the file. Then press one of the following keys:

  <RETURN> or <CURSOR-UP> or <CURSOR-DOWN>

  The cursor is positioned at the specified line regardless of whether this line is before or after the current

line.

<CURSOR-LEFT> or <CURSOR-RIGHT>

The window starts at the specified column. The cursor stays in the same line.

- relative to the current position:

  Enter a decimal number *n* preceded by a "+" or "–" sign and press one of the following keys:

<RETURN>	*n* lines down
<CURSOR-DOWN>	*n* lines down
<CURSOR-UP>	*n* lines up
<CURSOR-LEFT>	*n* columns to the left
<CURSOR-RIGHT>	*n* columns to the right

  It does not matter whether you enter *+n* or *–n*. The direction of movement is determined by the arrow keys only.

<MENU> x – Show programmed keys

The following text appears in the control panel:

`TO CONTINUE PRESS <MENU>`

In `x` mode you can display the keys that you have programmed and the texts/actions assigned to them. A complete list is displayed on the full screen, i.e. not in a `ced` window. The key is shown to the left of the first colon, and the string/action assigned to it is shown to the right.

The various symbols used, and their meanings are as follows:

->	<CURSOR-RIGHT>	cursor right
<-	<CURSOR-LEFT>	cursor left
UP	<CURSOR-UP>	cursor up
DOWN	<CURSOR-DOWN>	cursor down
HOME	<HOME>	cursor home
MENU	<MENU>	pressed
RETURN	<RETURN>	pressed

Keys <F9> through <F17>, which are predefined by `ced`, are not displayed. To return to the main menu, press <MENU>.

<MENU> ! – Call shell

The following text appears below the control panel:

```
>>> Exit to command shell, return to CED with the <CTRL-D> key.
```

This mode allows you to exit from `ced` temporarily. `ced` calls the program that you have specified in the shell variable `SHELL`. You can set this variable in your `.profile` file, for example. If `SHELL` is not set, it is not possible to call a shell. In this way you can work in the shell temporarily, without losing the contents of the edited file. You terminate the shell and return to `ced` by pressing <CTRL-D>.

Keys with special functions

In addition to the keys already described, the following keys have special functions:

<BACKSPACE>
: You can use this key to delete individual characters; also applies to the control panel.

<CTRL-R>
: You can use these keys to refresh the screen contents if they have been disturbed, e.g. by a message from a background process.

<CTRL-X>
These keys allow you to delete everything you have entered in the control panel.

<CTRL-@>
"emergency exit": if `ced` no longer responds to any input, you can terminate it with this key combination. Any files you have updated will not be saved.

If it is still not possible to input information normally, consult the system administrator.

Predefined standard functions

<START>
Go to start-of-file.

<F9> Save file.

<F10> Switch documents: change to a second file that you have already opened in `d` mode (switch documents) and back again.

<F11> Mark line: the cursor line is marked. It can then be copied with <F12>.

<F12> Copy line: a copy of the line that you previously marked with <F11> is inserted above the cursor line.

<F13> Search forward: you must previously have defined the required character string in `s` mode.

<F14> Search backward, otherwise same as <F13>.

<F15> Go forward one page.

<F16> Go back one page.

<F17> Retrieve line: the last line or line block that you deleted and buffered is retrieved and inserted (depending on whether you were in mode `l` or `b`).

Insert the labeled strip supplied with your machine (showing the predefined standard key functions) in the space provided on your keyboard.

ERROR MESSAGES

`ced: no status document /var/ced/CE`*`ttyname`*`.`*`loginname`*`, start`□
`not possible without document name`

- You tried to call `ced` without naming the file you want to edit. The file `/var/ced/CE`*`ttyname`*`.`*`loginname`* does not contain any suitable information on the file last edited. Call `ced` giving the name of the file you want to edit.

`ERROR IN COMMAND EXECUTION`

- You used an incorrect or unsuitable command in mode `c` (execute command).
- The syntax of the command you used is incorrect or invalid.

`INVALID INPUT`

- You pressed one of the arrow keys <HOME>, <CURSOR-LEFT> or <CURSOR-RIGHT> in mode `s` (search for string), where you can only use <CURSOR-UP>, <CURSOR-DOWN> and <RETURN>.
- You selected a non-existent function in mode `b` (edit line block), `l` (edit line), or `r` (edit rectangle).
- You attempted to choose a non-existent mode from the menu.
- You tried to exit the brief description of `ced` under `HELP` by repeatedly pressing <CTRL-D>. Exit `HELP` by pressing <MENU>.
- While in the selection menu you tried to terminate your `ced` session by pressing <CTRL-D>. Use mode `q` instead (quit `ced`)
- You tried to exit mode `s` (search for string) or `w` (position window) by pressing <CTRL-D>. Press <MENU> instead.

`invalid key: press <MENU> to continue`

You tried to exit mode `x` (display key assignments) by pressing <CTRL-D>. Press <MENU> instead.

`LINE TOO LONG`

The line you are editing has grown longer than the permissible 512 characters. This can happen if:

- you enter an overlong line in mode `e` (enter new text)
- you shift text to the right of the cursor beyond the 512th column in mode `i` (insert text).
- you use the `j` function in mode `l` (edit line), and the combined length of the two lines to be joined is greater than 512 characters.
- you use the `h` function in mode `r` (edit rectangle), and at least one line of the rectangle that you insert horizontally produces more than 512 characters when combined with the line already at that position.

`NO ALTERNATIVE DOCUMENT AVAILABLE`

When switching documents in mode `d` (switch documents), you did not give the name of the document you wanted to switch to and there are no other files open.

`NO TEXT`

You pressed function key <F13> (search forward) or <F14> (search backward) in mode `s` (search for string) without having defined a search string beforehand.

`NO TEXT AVAILABLE`

You marked a block in mode `b` (edit line block) and then attempted to retrieve it using the `i` function or the function key <F17> without first deleting and buffering it using the `d` function.

`THIS KEY CANNOT BE CHANGED`

You have made an attempt to program the <p> or <MENU> key in mode `p` (program keys). You should only use alphanumeric keys (but not p) or function keys <F1> through <F8> for macro assignments.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts and menu functions are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

ENVIRONMENT VARIABLES

`CED_TABS`
: If this variable is set to NO, spaces are not converted to tabs when a file is saved. However, initial tabs at the start of a line that were already in the file when the session began are retained.

`TMPDIR`
: By default, temporary files are stored in the `/tmp` directory.

  The `TMPDIR` variable defines the path under which `ced` stores temporary files if there is not enough space available in `/tmp`.

`SHELL` In this variable you define the command interpreter to be used when a shell is called or a command is executed. If this variable is not set, it is not possible to call a subshell or run a command from `ced`.

`TERM` This variable defines the type of terminal. You cannot start `ced` if this variable is not set.

FILES

`/var/ced/CE`*key*`.`*loginname*
: File in which your personal key assignments are stored.

`/var/ced/CE`*ttyname*`.`*loginname*
: File from which `ced` determines the file that you last edited with `ced` and the position at which the cursor was last located.

`/var/ced/CE`*error*`.`*number*
: File in which error messages generated during the execution of a Reliant UNIX command are temporarily stored.

`/tmp/CE`*textanumber*

File containing all changes made to a document in the course of a `ced` session.

`/tmp/CE`*textbnumber*
Copy of the file that is being edited with `ced`.

SEE ALSO

`ed`(1), `ex`(1), `sed`(1), `vi`(1).

checknr(1)
BSD Compatibility Package

NAME

`checknr` - check nroff and troff input files; report possible errors

SYNOPSIS

`/usr/ucb/checknr` [`-fs`] [`-a`*.x1.y1.x2.y2 ... .xn.yn*] [`-c`*.x1.x2.x3 ... .xn*] [*filename ...*]

DESCRIPTION

The `checknr` command checks a list of `nroff` or `troff` input files for certain kinds of errors involving mismatched opening and closing delimiters and unknown commands. If no files are specified, `checknr` checks the standard input. Delimiters checked are:

- Font changes using `\f`*x* ... `\fP`.
- Size changes using `\s`*x* ... `\s0`.
- Macros that come in open ... close forms, for example, the `.TS` and `.TE` macros which must always come in pairs.

`checknr` knows about the `ms` and `me` macro packages.

`checknr` is intended to be used on documents that are prepared with `checknr` in mind. It expects a certain document writing style for `\f` and `\s` commands, in that each `\f`*x* must be terminated with `\fP` and each `\s`*x* must be terminated with `\s0`. While it will work to directly go into the next font or explicitly specify the original font or point size, and many existing documents actually do this, such a practice will produce complaints from `checknr`. Since it is probably better to use the `\fP` and `\s0` forms anyway, you should think of this as a contribution to your document preparation style.

OPTIONS

`-f` Ignore `\f` font changes.

`-s` Ignore `\s` size changes.

`-a`*.x1.y1 ...*
Add pairs of macros to the list. The pairs of macros are assumed to be those (such as `.DS` and `.DE`) that should be checked for balance. The `-a` option must be followed by groups of six characters, each group defining a pair of macros. The six characters are a period, the first macro name, another period, and the second macro name. For example, to define a pair `.BS` and `.ES`, use "`-a.BS.ES`".

`-c`*.x1 ...*
Define commands (macros) which `checknr` would otherwise complain about

NOTE

There is no way to define a one-character macro name using the `-a` option.

SEE ALSO

`eqn`(1M), `nroff`(1M), `troff`(1M), `me`(5), `ms`(5).

User's Reference Manual

chgrp(1)

NAME

chgrp - change the group ownership of a file (change group)

SYNOPSIS

chgrp [–R] [–h] [––] *newgroup file* ...

DESCRIPTION

chgrp changes the user group for a file or a directory. You can only use it if you are the file/directory owner or the system administrator.

The system administrator has an unrestricted right to change the user group for any file.

The RSTCHOWN operating system configuration option can be used to place a restriction on the groups to which users without system administrator privileges can reassign their files. If this option is in effect, as an ordinary user you can assign your files to another group only if you are listed in the /etc/group file as a member of the new group (see **FILES**).

If chgrp is called by a user without system administrator privileges, any set-user-ID and set-group-ID bits [see chmod(1)] set for the specified files are cleared.

OPTIONS

–R (recursive) chgrp recursively descends through the specified directories, changing the group ID as it proceeds and traversing any symbolic links that it encounters.

–h If *file* is a symbolic link, chgrp changes the group ID of the symbolic link itself. Without this option, the group ID of the file referenced by the symbolic link is changed.

–– If *newgroup* begins with a dash (–), the end of the command-line options must be marked with ––.

newgroup
New group name or new group ID. *newgroup* must appear in /etc/group.

file Name of the file or directory for which the user group is to be redefined. You can also list any number of files and/or directories.

ERROR MESSAGES

file: Not owner

You are not permitted to change the user group of the specified file, since you are not the owner of the file or have not been entered as a member of the specified group or do not currently belong to the group. Only the system administrator is authorized to redefine the group for all files.

chgrp: unknown group: *newgroup*

The group name you have specified for *newgroup* is not in the /etc/group file.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

You are currently working under the login name cindy; this name is entered in the /etc/group file as a member of the user groups ag and prog. At present, you belong to the user group ag as is evident from the fact that the name ag is entered for "group" when you create new files.

```
$ >file
$ ls –l file
-rw-------   1 cindy    ag          0   Feb 17 15:48 file
```

You now wish to change the user group for `file`; the new group is to be `prog`. To do this, you first use the `newgrp` command to switch to the `prog` group and then change the group for `file` with `chgrp`.

```
$ newgrp prog
$ chgrp prog file
$ ls –l file
-rw-------   1 cindy    prog        0   Feb 17 15:48 file
```

FILES

`/etc/group`

The group file `/etc/group` contains a list of all existing user groups. Each line of this file consists of four colon-separated fields:

groupname : [*password*] : *groupid* : *user*,*user* ...

Only the system administrator is permitted to create new user groups and to enter new group members.

SEE ALSO

`chmod`(1), `chown`(1), `id`(1), `newgrp`(1), `chown`(2), `default_passwd`(4), `group`(4), `passwd`(4).

chmod(1)

NAME

`chmod` - change file mode

SYNOPSIS

`chmod` [–R] [––] *mode file ...*

DESCRIPTION

`chmod` is used to change the permissions of a file (its protection mode).

Only the owner or the system administrator is authorized to change the permissions of a file. A file's set-group-id bit can only be set by a user whose current group ID is the same as the file's group ID [see `chgrp(1)` and `newgrp(1)`].

OPTIONS

–R (recursive) `chmod` recursively descends through the specified directories, changing the mode for each file it encounters.

–– If *mode* begins with a dash (–), the end of the command line options must be marked with ––.

mode In *mode* you specify how the permissions for one or more named files are to be changed. There are two forms of *mode* specification:

- symbolic
- absolute

Symbolic form: [*who*]*op*[*permission*][,[*who*]*op*[*permission*]]...

who In *who* you say who the file permissions apply to.

The choices for *who* are:

`u` for the owner (user)

`g` for the group

`o` for others

`a` for `ugo`, i.e. all users.

or any combination of the letters `u`, `g`, `o`.

who not specified:

`chmod` behaves in the same way as if `ugo` or `a` had been specified. The access rights are changed for owner, group, and other, with the exception of the bits protected by `umask(1)` (see **EXAMPLES**, 2nd table).

op In *op* you specify whether permissions are to be granted, left unchanged, or revoked.

The choices for *op* are:

`+` to add (grant) permissions,

`–` to take away (revoke) permissions,

`=` to assign permissions absolutely, i.e. only the listed permissions are granted are granted, all others are revoked.

permission

In *permission* you specify which permission(s) you wish to grant or revoke.

The choices for *permission* are:

`r` for read permission

`w` for write permission

`x` for execute permission or for permission to search in a directory.

`X` (X – execute) for execute permission or for permission to search in directories. `X` only works in opposition to `x` in the case of directories or if `x` access permissions are already set for *file*.

`s` for the set-user-ID or set-group-ID bit.

Entering `s` in a `chmod` command is only useful in conjunction with `u`, `g`, or `ug` (if *who* is not specified, it defaults to `ug`). Set-user-ID and set-group-ID bits only apply to executable binary files (not to shell scripts) (see **The s bits** below).

`t` for sticky bit (t bit).

Only the system administrator is able to set the sticky bit. Attempts by non-privileged users to set the sticky bit are ignored. Entering `t` in a `chmod` command is useful only in combination with `u` or `a` or if *who* is not defined. A set sticky bit only applies to executable files (see **The sticky bit** below). If you change the mode of a file which has the sticky bit set, the sticky bit is cleared automatically.

`l` (lock) for mandatory locking of files, directories or records, referring to a file's ability to have its read or write permissions locked while a program is accessing it. *file*'s `l` bit can be set only if

- its group execute permission is not set and
- its set-group-id bit is set.

Thus the following examples are **not** correct and would result in error messages:

```
chmod g+x,+l file
chmod g+s,+l file
```

`u` Use the permissions of the current owner.

`g` Use the permissions of the current group.

`o` Use the permissions of the current "others" mode.

permission not specified:

This is only useful in combination with the `=` operator; all permissions are then revoked for the *who* in question.

As indicated above, you can specify several "who-op-permission" arguments in a row, provided they are separated by commas as follows:

```
chmod g-w,o-rw file
```

The string you specify for *mode* is processed by `chmod` from left to right. For instance, `a-w,u+w` grants write permission to the owner, revoking it for all others.

Absolute form

An absolute *mode* is a three or four digit octal number. The admissible octal values are obtained by logically ORing (in binary) the octal modes shown below. The effect is the same as adding the modes in octal or decimal. A leading zero (neither the s bit nor the sticky bit set) may be omitted.

The specified permissions are granted; all other permissions are revoked.

4000 Set user ID on execution

20#0 Set group ID on execution if # is 7, 5, 3 or 1. Set mandatory locking if # is 6, 4, 2 or 0. The value of # is ignored if *file* is a directory. In this case you can only use the symbolic form for *mode*.

1000 sticky bit (t bit)

0400 read permission for owner

0200 write permission for owner

0100 execute permission (or directory search permission) for owner

0040 read permission for group

0020 write permission for group

0010 execute permission (or directory search permission) for group

0004 read permission for others

0002 write permission for others

0001 execute permission (or directory search permission) for others

Example:

To grant read, write, and execute permission to the owner and read and execute permission to the group, you would enter a value of 750 for *mode*:

400 + 200 + 100 + 40 + 10 = 750

The file then has the permissions `rwxr-x---`

file Name of the file for which you wish to define or alter the permissions. *file* may also be a directory. Multiple names are also allowed.

SPECIAL PERMISSIONS

The s bit

When an executable program which has the set-user-ID bit set is called, the *effective* user ID of the associated process is the same as the user ID of the *owner* of the file (and not that of the caller). In other words, the process runs under the user ID of the program owner and can therefore also access files for which the caller of the program does not have explicit access permission. The *real* user ID of the process remains that of the program *caller*.

If the set-group-ID bit is set, the effective group ID of the process is the same as the group ID of the program *owner*. This means that the process runs under the group ID of the program owner. The *real* group ID of the process remains that of the program *caller*.

The s bits are only useful for executable binary files (executable programs) and not for shell scripts. Although `chmod` can be used to set the s bits for files that contain a shell script, the setting will essentially have no effect.

When changes are made to a file, the s bits are reset for security reasons.

Caution:

If the system administrator sets the s bit for a program which he or she owns, all users who can call the program are thereby authorized to carry out any operations that the system administrator is allowed to perform using the program. Hence the system administrator should not set the s bit unless it is certain that security will not be compromised, leading to loss of data, for example.

One example of an s bit application is the `passwd` command (`/bin` directory). As a normal user, you are not granted write permission for the password file `/etc/passwd`. However, since the set-user-ID bit for `root` has been set for the `passwd` command, you can use this command to give yourself a new password in the `/etc/passwd` file.

The sticky bit (t bit)

Only the system administrator can set the sticky bit (t bit). Attempts by non-privileged users to set the sticky bit are ignored.

The sticky bit only works on directories and executable files. It is possible to use `chmod` to set it on other files, but it will have no effect.

If the sticky bit is set on an executable file, the overhead typically involved in reading a program in from the disk file again every time the program is started can be reduced somewhat.

If a directory is write enabled and has the sticky bit set, files within the directory cannot be removed, renamed or linked to unless one or more of the following conditions apply:

- the file belongs to the user
- the directory belongs to the user

- the user has write permission for the file
- the user is a privileged user

In the output of the `ls -l` command, the sticky bit if set appears in the last position of the listed permissions. If the x bit has simultaneously been set for "other users", a `t` appears; if not, a `T`.

The l bit

The `lockf()` function allows a program to place a lock on a file which it is accessing. If this file has its l bit set, the function call results in mandatory locking of the file [see `lockf`(3C)].

ERROR MESSAGES

`chmod: ERROR: Invalid mode`

You have defined an illegal set of permissions for `chmod`.

`chmod: WARNING: Locking not permitted on` *file*`, a group executable file`

Files with group execute permission cannot also have the l bit set.

`chmod: WARNING: Execute permission required for set-ID on execution for`*file*

In order to turn on a file's set-user-ID bit you need to have execute permission for the file.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The first table shows examples of which symbolic entries correspond to which absolute entries.

Symbolic form	Absolute form
`u-w`	400
`-w`	400
`+rx`	755
`=r`	444
`ug=rw,o=r`	664
`u=rwx,g=rx,o=`	750
`+x,u+s`	4711
`+xt`	1711

Note: The sticky bit (last example) can only be set by the system administrator. Attempts by non-privileged users to set the sticky bit are ignored.

The following examples all refer to a file with the permissions `rw-------`.

The second table shows the different results with symbolic entries without *who* being specified, depending on which file-creation mode mask is set with `umask`(1) .

Symbolic form	`umask`	Result
`+w`	022	`rw-------`

+w	000	rw--w--w-
=rw	022	rw-r--r--
=rw	000	rw-rw-rw-

The third table shows results of symbolic or absolute entries where the file-creation mode mask set with umask(1) does not play a role.

Symbolic form	Absolute form	umask	Result
go+r	644	_	rw-r--r--
go=r	644	_	rw-r--r--
go+rw	666	_	rw-rw-rw-

SEE ALSO

chgrp(1), ls(1), newgrp(1), umask(1), chmod(2), chown(2), lockf(3C), passwd(4).

User's Reference Manual

chown(1)

NAME

`chown` - change file owner

SYNOPSIS

`chown` [–R] [–h] [––] *newowner*[:*newgroup*] *file* ...

DESCRIPTION

`chown` assigns a new owner and, where necessary, a new group to a file or a directory. The group is only changed if *newgroup* is specified explicitly.

The system administrator has an unrestricted right to change the owner of any file.

By contrast, normal users without system administrator privileges may only change the owner ID of their own files. If the operating system option `RSTCHOWN` is in effect, even the owner of a file is prevented from changing the ownership of that file. In such a case, an error message `chown: `*file*`: Not owner` (see **ERROR MESSAGES**) is displayed.

If `chown` is invoked by a user other than the system administrator, the set-user-ID bit of the file mode, 4000, is cleared.

OPTIONS

`–R` (recursive) `chown` recursively descends through the specified directories, changing the file owner as it proceeds and traversing any symbolic links that it encounters.

`–h` If *file* is a symbolic link, `chown` changes the owner of the symbolic link. Without this option, the owner of the file referenced by the symbolic link is changed.

`––` If *newowner* begins with a dash (–), the end of the command-line options must be marked with ––.

newowner
: Login name or user ID of the new owner. Must be one that is entered in the `/etc/passwd` file.

newgroup
: New group name or new group number. *newgroup* must be entered in the `/etc/group` file.

file Name of the file that is to have a new owner. Directories and multiple file names are also allowed.

ERROR MESSAGES

`chown: `*file*`: Not owner`

You are not permitted to change the owner of the named *file*, as you are not working as the system administrator or the `RSTCHOWN` variable is set on your system.

`chown: unknown user id` *newowner*

The login name you specified for *newowner* is not entered in the `/etc/passwd` file.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

As system administrator you want to change the owner of a file; the owner of the file `text1`, currently user `cathy`, is to be changed to `mark`, while the owner of the file `text2` is to be changed to `fred`. To do this, you enter:

```
# ls –l text1 text2
-rw-------   1 cathy    ag          2426   Feb 17 15:48 text1
-rw-------   1 cathy    ag          1587   Feb 17 15:49 text2
# chown mark text1
# ls –l text1
-rw-------   1 mark     ag          2426   Feb 17 15:48 text1
# chown fred:usrother text2
# ls -l text2
-rw-------   1 fred     usrother    1587   Feb 17 15:49 text2
```

EXIT STATUS

`0` The utility executed successfully and all requested changes were made.

`>0` An error occurred.

FILES

`/etc/passwd`
: The `/etc/passwd` file contains information on all existing users.

  Only the system administrator is allowed to add new users.

`/etc/group`
: The `/etc/group` file contains all user groups created.

  Only the system administrator can create new user groups and enter new group members.

SEE ALSO

`chgrp`(1), `chmod`(1), `chown`(2), `passwd`(4).

User's Reference Manual

chown(1-ucb)
BSD Compatibility Package

NAME

`chown` - change file owner

SYNOPSIS

`/usr/ucb/chown` [`–fhR`] *owner*[*.group*] *file* ...

DESCRIPTION

`chown` changes the owner of the *file*s to *owner*. The *owner* may be either a decimal user ID or a login name found in `/etc/passwd` file. The optional *.group* suffix may be used to change the group at the same time.

If `chown` is invoked by other than the super-user, the set-user-ID bit of the file mode, 04000, is cleared.

Only the super-user may change the owner of a file.

OPTIONS

`–f` Suppress error reporting.

`–h` If the file is a symbolic link, change the owner of the symbolic link. Without this option, the owner of the file referenced by the symbolic link is changed.

`–R` Descend recursively through directories setting the ownership ID of all files in each directory entered.

NOTES

In a remote file sharing environment, you may not have the permissions that the output of the `ls –l` command leads you to believe.

FILES

`/etc/group`
`/etc/passwd`

SEE ALSO

`chgrp`(1) , `chmod`(1) , `chown`(2) , `passwd`(4) .

User's Reference Manual

chsh(1)

NAME

`chsh` - change default login shell

SYNOPSIS

`/usr/ucb/chsh` *name* [*shell*]

DESCRIPTION

`chsh` is a command similar to `passwd(1)`, except that it is used to change the login shell field of the password file rather than the password entry.

If no *shell* is specified, or the standard login shell `/sbin/sh` is specified, no shell will be entered in `/etc/passwd`. The standard shell will be used the next time the user logs in.

You can only specify

- `/bin/csh`
- `/bin/ksh`
- `/bin/sh`
- `/usr/bin/csh`
- `/usr/bin/ksh`
- `/usr/bin/sh`

as the *shell*, unless you are the super-user.

An example use of this command would be

```
chsh bill /bin/csh
```

If a field in the `/etc/passwd` file is corrupt, processing will be terminated without copying the rest of the passwd file. If this happens, a message is printed and the system administrator should be contacted. The administrator can look at the temporary file `/etc/ptmp` to see what the last good record was, fix the bad record in `/etc/opasswd`, and copy it to `/etc/passwd`.

SEE ALSO

`passwd(1)`, `passmgmt(1M)`, `passwd(4)`.

User's Reference Manual

cksum(1)

NAME

`cksum` - write file checksums and sizes

SYNOPSIS

`cksum` [`-C`] [*file* ...]

DESCRIPTION

The `cksum` utility calculates and writes to standard output a cyclic redundancy check (CRC) for each input file, and also writes to standard output the number of octets in each file. The CRC used is based on the polynomial used for CRC error checking in the referenced Ethernet standard.

The encoding for the CRC checksum is defined by the generating polynominal:

$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

Mathematically, the CRC value corresponding to a given file is defined by the following procedure:

1. The n bits to be evaluated are considered to be the coefficients of a mod 2 polynomial $M(x)$ of degree n-1. These n bits are the bits from the file, with the most significant bit being the most significant bit of the first octet of the file and the last bit being the least significant bit of the last octet, padded with zero bits (if necessary) to achieve an integral number of octets, followed by one or more octets representing the length of the file as a binary value, least significant octet first. The smallest number of octets capable of representing this integer is used.
2. $M(x)$ is multiplied by x^{32} (that is, shifted left 32 bits) and divided by $G(x)$ using mod 2 division, producing a remainder $R(x)$ of degree ≤ 31.
3. The coefficients of $R(x)$ are considered to be a 32-bit sequence.
4. The bit sequence is complemented and the result is the CRC.

OPTIONS

`-C` calculates the CRC checksum in the same way as in previous versions.

file A pathname of a file to be checked. If no file operands are specified, the standard input is used.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of `cksum`:

`LANG` Provide a default value for the internationalization variables that are unset or null. If `LANG` is unset of null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalization variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

`LC_ALL`
If set to a non-empty string value, override the values of all the other internationalization variables.

`LC_CTYPE`
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

`LC_MESSAGES`
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

`NLSPATH`
Determine the location of message catalogues for the processing of `LC_MESSAGES`.

STDOUT

For each file processed successfully, the `cksum` utility will write in the following format:

`"%u %d %s\n", checksum, #_of_octets, pathname`

If no file operand was specified, the pathname and its leading space will be omitted.

EXIT STATUS

The following exit values are returned:

`0` All files were processed successfully.

`>0` An error occurred.

APPLICATION USAGE

The `cksum` utility is typically used to quickly compare a suspect file against a trusted version of the same, such as to ensure that files transmitted over noisy media arrive intact. However, this comparison cannot be considered cryptographically secure. The chances of a damaged file producing the same CRC as the original are astronomically small; deliberate deception is difficult, but probably not impossible.

Although input files to `cksum` can be any type, the results need not be what would be expected on character special device files or on file types not described by the XSH specification. Since this document does not specify the block size used when doing input, checksums of character special files need not process all of the data in those files.

The algorithm is expressed in terms of a bitstream divided into octets. If a file is transmitted between two systems and undergoes any data transformation (such as moving 8-bit characters into 9-bit bytes or changing "Little Endian"), identical CRC values cannot be expected. Implementations performing such transformations may extend `cksum` to handle such situations.

NOTES

The following error message is displayed if *file* is the mount point for an NFS file system:

`cksum: Read error on file: Is a directory`

SEE ALSO

`sum(1)` .

User's Reference Manual

clear(1)

NAME

`clear -` clear the terminal screen

SYNOPSIS

```
clear
```

DESCRIPTION

`clear` clears the terminal screen. Once the screen is clear, the shell displays the prompt.

`clear` determines the terminal type from the environment variable `TERM`.

`clear` uses this information to decide whether and how to clear the screen. If the `TERM` variable is set, the screen is cleared.

ENVIRONMENT VARIABLES

`TERM` Contains information on the terminal type.

EXAMPLES

```
$ set | grep TERM
TERM=98765
$ clear
$
```

SEE ALSO

`tput(1)`.

User's Reference Manual

cmp(1)

NAME

cmp - compare two files

SYNOPSIS

cmp [-l] [-s] [--] *file1 file2*

DESCRIPTION

The cmp command does a comparison of two files byte by byte (character by character). If the files differ, cmp reports the differences on standard output.

If the files are identical, cmp remains silent.

OPTIONS

No option specified:

If the files are identical, cmp remains silent.

If the files differ, cmp indicates the byte (character) and line number of the first difference that it detects between *file1* and *file2* as shown below:

file1 file2 differ: char *bytenumber*, line *linenumber*

-l All differences are reported in the following form:

bytenumber byte(file1) byte(file2)

bytenumber represents the displacement of the difference from the beginning of the file. The first byte of the file is assigned number 1 and blanks are counted.

bytenumber is given in decimal notation.

The *byte* columns show the bytes which differ between *file1* and *file2* and are in octal notation. An ASCII table of octal-coded values is provided in ascii(5) . If the files are identical, nothing is output.

-s cmp remains silent. The exit status value is returned but not automatically displayed on the screen. -s must not be used together with -l.

-- If *file1* begins with a dash (-), the end of the command-line options must be marked with --.

file1 file2

The names of the files that you wish to compare.

- If you use a dash - as the name for *file1*, cmp reads from standard input and compares your input with *file2*.
- If one of the two files ends before cmp can detect a difference, cmp reports that the end of the file has been reached in the shorter file by issuing the following message:

  cmp: EOF on *file*
- If one character is missing in one of two otherwise identical files, cmp -l reports all following bytes as differences. This is due to the shift in character positions.
- The first character of a file is assigned *bytenumber* 1, not 0.
- Blanks and newline characters are included in the *bytenumber* count.

EXIT STATUS

0 Files identical.

1 Files differ.

>1 Inaccessible file or missing argument.

ERROR MESSAGES

`cmp: cannot open` *file*

You do not have read permission for one of the files, or one of the files does not exist.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Compare two files and print the differing *bytes* (octal) and their *bytenumber* (decimal).

```
$ echo 1 2 3 4 5 7 8 a >file1
$ echo 1 2 3 4 5 6 9 a >file2
$ cmp –l file1 file2
11  67  66
13  70  71
$
```

Example 2

The shell script `delete.eq` compares two files and deletes one of them if they are identical.

```
if cmp –s $1 $2
then
  rm $2
fi
```

When you call the script with

```
$ delete.eq file1 file2
```

you pass `file1` and `file2` to it as positional operands. The `–s` option causes `cmp` to return the exit status. If the value of the exit status is 0 (= true), `file2` is deleted; otherwise, it is retained.

SEE ALSO

`comm(1)`, `diff(1)`.

User's Reference Manual

col(1)

NAME

`col` - filter reverse line feeds

SYNOPSIS

`col` [*option* ...]

DESCRIPTION

`col` reads from the standard input and writes the result to standard output. It performs the line overlays implied by reverse line feeds and forward or reverse half-line feeds.

`col` accepts half-line motions as input, but does not normally transfer them to standard output. Instead, text that would appear between lines is moved to before or after the next lower full-line boundary. Reverse line feeds starting from the first line of a file are ignored.

On output, `col` substitutes tabs for blanks (octal 040), provided the position of the following non-blank character is not affected. This shortens printing time. The default spacing between tab stops is 8 characters. The ASCII control characters `SO` and `SI` [see `ascii(5)`] are assumed by `col` to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output `SI` and `SO` characters are generated as appropriate to ensure that each character is printed in the correct character set.

`col` is particularly useful for filtering multicolumn output produced by the `.rt` command in `nroff(1M)` and output resulting from use of the `tbl(1M)` preprocessor. The input format accepted by `col` matches the output produced by `nroff` with either the `–T37` or `–Tlp` options. You should use `–T37` (and the `–f` option of `col`) if the ultimate destination of the output of `col` is a device that can interpret half-line feeds, and `–Tlp` otherwise.

`col` ignores local vertical motions that would result in backing up over the first line of the document. Thus the first line must not have any superscripts.

`col` cannot back up more than 128 lines.

Non-printing and non-ASCII characters in the input are ignored, with the following exceptions:

Non-printing characters interpreted by `col`	**ASCII value (octal)**	**entered as**
Space (SP)	040	space bar
Backspace (BS)	010	BACKSPACE
Horizontal Tab (HT)	011	TAB
Carriage Return (CR)	015	CTRL-M or appropriate key
Line Feed (LF)	012	CTRL-J or appropriate key
Shift In (SI)	017	CTRL-O
Shift Out (SO)	016	CTRL-N
Vertical Tab (VT)	013	CTRL-K
Reverse line feed	033 067	ESC 7
Reverse half-line feed	033 070	ESC 8
Forward half-line feed	033 071	ESC 9

The CTRL key and the second key have to be pressed simultaneously; the ESC key (033) and the second key have to be pressed consecutively.

Unless the –p option is set, col normally removes any escape sequences found in its input that are unknown to it.

OPTIONS

–b col assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

–f col outputs forward half-line feeds, but does not output either kind of reverse line motion (half-line or full-line).

–p col will also accept escape sequences other than the ones described above. Use –p only if you are fully aware of the consequences!

–x col does not convert white space in input to tabs on output.

ERROR MESSAGES

```
col: bad option
```

You have specified an invalid option.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file texts contains the following characters:

```
text1 ESC 9text2 ESC 9text3 ESC 9text4 ESC 8 ESC 8text5
```

The following screen output is generated by col:

```
$ cat texts | col
text1
      text2text3     text5
                text4
```

The contents of a file containing these characters could also be output in this way with normal forward processing.

SEE ALSO

nroff(1M), tbl(1M), ascii(5), meta-ascii(5).

User's Reference Manual

colon(1)

NAME

`colon, :`

`-` return zero exit status

SYNOPSIS

`:` [*argument* ...]

DESCRIPTION

The shell built-in `:` (colon) returns zero exit status and does nothing else. It is used in shell scripts in the following way:

- Without command-line arguments, it does exactly the same as the `true(1)` command. This means that you can also create the condition "true" using the shell built-in `:`.
- If you include arguments, the shell interprets all metacharacters that appear in them. This method of using the shell built-in `:` enables you to assign a default value to free shell variables without initiating any other action [see `sh(1)` , section **Defining default values for shell parameters**].
- If you do not use any shell metacharacters in the command-line arguments, or if you escape them, the shell built-in `:` has the same function as the hash character #, i.e. it introduces comments. With `:`, though, unlike #, the next unescaped command separator terminates the comment.

ARGUMENTS

argument

Any string delimited by blanks or tabs. The last argument must be terminated by a command separator.

You can specify any number of arguments, provided they are separated by at least one tab or blank.

As with every other command, the string is first interpreted by the shell [see `sh(1)`]. The shell built-in `:` returns only a zero exit status.

argument not specified:

The shell built-in `:` returns only a zero exit status and does nothing else.

EXIT STATUS

`0` in every case

EXAMPLES

Example 1:

You can use the shell built-in `:` to fill in a branch of an `if` or `case` construct if you do not want anything to happen in that branch. Suppose the shell script `not_x` contains the following:

```
if test -x $1
then :
else echo $1 is not executable!
fi
```

This shell script tests the file that you name as your first command-line argument. If the file is executable, the shell script does nothing; otherwise, the message "*file* is not executable!" is issued.

Example 2:

The following shell script assigns the process ID of the current shell process to the shell variable `name` if this variable is undefined or contains the null string:

```
: ${name:=$$}
echo $name
```

SEE ALSO

ksh(1), sh(1), true(1).

User's Reference Manual

comm(1)

NAME

comm - select or reject lines common to two sorted files

SYNOPSIS

comm [*option*] *file1 file2*

DESCRIPTION

comm compares two files in which the lines are sorted on the basis of the currently valid collating sequence. Sorting can be performed with the sort command [see sort(1)].

OPTIONS

No options specified:

comm produces three columns with the following meanings:

Column 1	Column 2	Column 3
lines which occur in *file1* only	lines which occur in *file2* only	lines which occur in both files

−1 Column 1 is not output.

−2 Column 2 is not output.

−3 Column 3 is not output.

−12 comm outputs all lines common to both files.

−23 comm outputs all lines which only occur in *file1*

−13 comm outputs all lines which only occur in *file2*

−123 comm generates no output.

−− If *file1* begins with a dash (−), the end of the command line options must be marked with −−.

file1 file2

Names of the two sorted files which you want to compare.

The comm command will not function properly unless both files have been sorted. If you use a dash as one of the names, comm reads from standard input.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

The LC_COLLATE environment variable governs the collating sequence.

If LC_MESSAGES or LC_COLLATE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file books contains the titles of books and their authors. Each line contains the title of one book and the name of its author, with a space between them. You would now like to search the file books for a number of authors whose names you have listed in the file authors1. The contents of books and authors1 are as follows:

```
books                    authors1
"Gormenghast" Peake      Blyton
"Buddenbrooks" Mann      Gogol
"Noddy" Blyton           Joyce
"Ulysses" Joyce          Kafka
                         Mann
                         Tolstoy
```

You can now proceed as follows:

- Use `awk` to extract the authors from `books`.
- Sort the authors in `books` using `sort`.
- Redirect the output of `sort` to the new file `authors2`.
- Compare the files `authors1` and `authors2` using `comm -2`.

```
$ awk '{printf"%s\n",$2}'books | sort > authors2
```

The file `authors2` contains the following:

```
Blyton
Joyce
Mann
Peake
$ comm -2 authors1 authors2
         Blyton
Gogol
         Joyce
Kafka
         Mann
Tolstoy
```

All authors who only appear in `authors1` are output in column 1, followed by the contents of column 3, which lists all authors present in both files.

SEE ALSO

`cmp(1)`, `diff(1)`, `sort(1)`, `uniq(1)`.

User's Reference Manual

command(1)

NAME

`command` - execute a simple command

SYNOPSIS

`command` [`-p`] *commandname* [*argument* ...]

`command` [`-v` | `-V`] *commandname*

DESCRIPTION

`command` causes the shell to treat the arguments as a simple command, suppressing the shell function lookup.

If the *commandname* is the same as the name of one of the special built-in commands, the special properties of the special built-in command will not occur. In every other respect, if *commandname* is not the name of a function, the effect of `command` will be the same as omitting `command`.

`command` also provides information as to how a command name is interpreted by the shell. See `-v` and `-V`.

OPTIONS

`-p` Performs the command search using a default value for `PATH`. All of the standard commands can be found using these default pathnames.

`-v` Writes the pathname or command, with which the shell invokes *commandname* in its current execution environment, to standard output.

- Commands, regular built-in commands, command names (*commandname*) including a slash character, and all available functions that can be found using the `PATH` variable, are written as absolute pathnames.
- Shell functions, special built-in commands, and regular built-in commands that cannot be found using `PATH`, as well as reserved words within the shell are written as just simple names.
- An alias will be written as the alias definition.
- In all other cases, no output is written, and the exit status will show that the name was not found.

`-V` Specifies how *commandname* is interpreted by the shell in its current execution environment. The output specifies which of the categories listed below *commandname* belongs to, and contains the following information:

- Commands, regular built-in commands, and all available functions that can be found using the `PATH` variable are identified as such and are written with absolute pathnames.
- Other shell functions are identified as functions.
- Aliases are identified as such, and are written as alias definitions.
- Special built-in commands are identified as such.
- Regular built-in commands that cannot be found using `PATH` are identified as regular built-in commands.
- Reserved words in the shell are identified as reserved words.

argument
Character string treated as an argument to *commandname*.

commandname
The name of a command or a special built-in command.

ENVIRONMENT VARIABLES

`LANG` Provides a default value for the internationalization variables that are unset or null. If `LANG` is unset or

null, the corresponding default value from the internationalized environment is used. If one of the internationalization variables contains an invalid setting, the command behaves as if none of the variables has been defined.

`LC_ALL`
If this variable has been assigned a value, i.e. it is not a null string, this value overrides the values of all the other internationalization variables.

`LC_CTYPE`
Determines the internationalized environment for the interpretation of byte sequences as characters (e.g. single-byte characters as opposed to multi-byte characters in arguments).

`LC_MESSAGES`
Determines the format and contents of error messages. The internationalized environment specified here is also valid for informative messages written to standard output.

`NLSPATH`
Determines the position of message catalogs for processing `LC_MESSAGES`.

`PATH` Determines the search path used during the command search, except where a default value has been set with `-p`.

EXIT STATUS

If one of the options `-v` or `-V` is specified, the following exit values are returned:

`0` Successful completion.

`>0` *commandname* could not be invoked or an error occurred.

Otherwise, the following exit values are returned:

`126` The command specified by *commandname* was found, but could not be invoked.

`127` An error occurred in the `command` or the command specified by *commandname* could not be found.

Otherwise, the exit status of *commandname* is returned.

APPLICATION USAGE

The order for a command search allows functions to override regular built-in utilities and search paths. `command` allows functions that have the same name as a command to invoke the command (instead of a recursive call to the function).

The system default path can be queried using `getconf(1)` . Since `getconf` may need to be invoked itself using `PATH`, the following call can be used:

```
command -p getconf _CS_PATH
```

There are some advantages to suppressing the special characteristics of special built-in commands. In the example below, the abort of a non-interactive script is suppressed, so that the output status of the script can be checked.

```
command exec > unwritable-file
```

Since the `-v` and `-V` options of `command` produce output in relation to the current shell execution environment, `command` is provided as a regular built-in shell command. If it is called in a subshell or a separate execution environment, as in the example below, it may produce incorrect results:

```
(PATH=foo command -v)
nohup command -v
```

If, for example, it is called with `nohup` or `exec` in a separate execution environment, most implementations will not be able to identify aliases, functions or special built-in commands.

EXAMPLES

1. A version of `cd` that outputs the new working directory can be made as follows:

```
cd1() {
 command cd "$@" >/dev/null
```

```
 pwd
}
```

Note: Since `cd` is a built-in shell command, it must not be redefined according to the X/Open specification. For this reason, the name "cd" must be changed to "cd1" [cf. sh(1), section **Defining a shell function**].

2. A "secure shell script" that cannot be spoofed by its parent process can be started as follows:

```
IFS='
'
# the preceding value should be <space><tab><newline>.
# set IFS to its default value.
\unalias -a
# Unset all possible aliases.
# Note that unalias is escaped to prevent an alias
# being used for unalias.
unset -f command
# Ensure command is not a user function.
PATH="$(command -p getconf _CS_PATH):$PATH"
# Put on a reliable PATH prefix.
# ...
```

Provided that the correct permissions exist for the directories called by `PATH`, the script can now ensure that any command it invokes is the intended one.

NOTES

`command` exists both as an external command (`/usr/bin/command`) and as a built-in shell command in the Korn shell ksh(1). The shell generates a new process to execute `/usr/bin/command`.

Some differences in behavior may occur when using `command`, depending on which command is being used. The possible differences are not described specifically.

SEE ALSO

getconf(1), ksh(1), type(1).

User's Reference Manual

compress(1)

NAME

`compress` - compress files

SYNOPSIS

`compress` [*option ...*] [*file ...*]

DESCRIPTION

`compress` reduces the size of files using adaptive Lempel-Ziv coding: recurrent strings in the text are reduced to unique codes ranging from 9 to a maximum of 16 bits.

On systems which evaluate the `RSTCHOWN` variable, the user who invokes `compress` is made the owner of the compressed file (unless the invoking user is the system administrator). On other systems the owner stays the same.

The access permissions, and the dates of last access and modification stay the same. Each file is replaced by a file with the same basename plus a `.Z` extension.

The amount of compression obtained depends on the size of the input file, the value of *maxbits* (see option `-b` below), and the distribution of recurrent substrings. Typically, files containing only text or source code are reduced by 50-60%. Compression achieved with the Lempel-Ziv approach is generally much better than that achieved with Huffman coding [see `pack(1)`], and takes less time to compute.

No compression is performed if

- the input file is not a regular file (if it is a directory, for example)
- there are other links to the input file
- the input file already has a `.Z` extension
- the `.Z` file to be created already exists, and `compress` is being executed in the background (`/usr/bin/sh`)
- no savings can be achieved by compression

Files compressed with `compress` can be decompressed with `uncompress(1)`.

`zcat(1)` produces uncompressed output on the standard output, but leaves the compressed file intact.

OPTIONS

No option specified:

The specified files are compressed if storage space can be saved as a result.

`-c` Simply writes the output of `compress` to standard output without modifying or creating any files. Only one file may be specified for `-c`.

`-f` Forces compression even if no space savings will be achieved or the `.Z` file that will be created already exists. If the `.Z` file already exists it will be overwritten.

`-f` not specified:

`compress` prompts to verify whether an existing `.Z` file should be overwritten. No prompt is issued if `compress` is running in the background.

`-v` (verbose) Displays the percentage reduction for each file compressed:

file `Compression: xx.xx% -- replaced with` *file*`.Z`

`-b` *maxbits*

Sets the maximum size (in bits) for recurrent substring codes to *maxbits*. The value for *maxbits* must be between 9 and 16. Lowering the number of bits will result in larger, less compressed files.

The *maxbits* parameter specified during compression is encoded within the compressed file, together with a magic number to ensure that it is impossible to further compress a file which is already

compressed.

–b not specified: The value of *maxbits* defaults to 16.

Caution:

Although compressed files are compatible between machines with large memory, the –b 12 option should be used for file transfer to architectures with a small process data space (64 Kb or less).

–– If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file Name of the file to be compressed. You may specify more than one file. If you specified a dash – for one of the files, compress reads from the standard input at this point.

file must not be a directory and there must be no other links to it.

The compressed file is assigned the name *file*.Z, and the named *file* is deleted after successful compression. *file*.Z has the same access permissions, access time and modification time as *file*. The maximum length of the name of *file* depends on the file system being used. The maximum allowable length is equal to the maximum file name length supported by the file system, minus 2 characters. This allows for the extension to *file*.Z. If this length is exceeded, *file* will not be compressed. No .Z files are created for the standard input, the data is written in compressed form to standard output.

Caution:

If compress is running in the background, an existing .Z file is overwritten without a prompt.

file not specified:

Data from standard input is written in compressed form to standard output.

EXIT STATUS

0 Compression successful

1 Error

2 Compression not performed, as it would have increased the size of one or more files. (Can only occur if –f was not specified.)

>2 Error

ERROR MESSAGES

file: filename too long to tack on .Z
The name of the compressed file is too long. No compression takes place.

file -- not a regular file: unchanged
If the input file is not a regular file it is left uncompressed.

file: -- has *xx* other links: unchanged
The input file has *xx* other links; it is left uncompressed.

file unchanged
No savings can be achieved by compression. The input file remains uncompressed. The –f option can be used to force compression.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

Answers to yes/no queries must be given in the language appropriate to the current locale.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file films, which occupies 4862 bytes in uncompressed form, is to be compressed.

```
$ ls –l
total 10
-rw-------  1  felix    group1     4862 Aug 19 09:27 films
$ compress –v films
compress: films: 50.78% Compression -- replaced with films.Z
$ ls –l
total 6
-rw-------  1  felix    group1     2393 Aug 19 09:27 films.Z
```

SEE ALSO

pack(1), pcat(1), uncompress(1), unpack(1), zcat(1).

User's Reference Manual

copy(1)

NAME

`copy` - copy groups of files

SYNOPSIS

`copy` [*–option ...*] *source ... destination*

DESCRIPTION

The `copy` command can be used to copy the contents of one or more directories into another directory or to copy user and special files.

OPTIONS

Any number of option arguments may be entered in any order.

No option specified:

If *source* is not a directory, `copy` is identical to the `cp` command.

If *source* is a directory, all files in the source directory are copied. Subdirectories are not taken into account.

If the named destination (file or directory) does not exist, a new one is created and is given the same attributes as the source. In the case of `copy`, as opposed to `cp`, there are no restrictions placed on the access permissions as a result of the `umask` setting.

If the named destination does exist it is overwritten.

`–a` Before copying each file, `copy` requests you to individually confirm whether or not a copy is to be made. The command will only be executed if you answer `y`.

The `–a` option automatically sets the `–ad` option.

`–ad` `copy` requests confirmation before copying a subdirectory. The command will only be executed if you answer `y`.

`–l` (link) For files, links are made instead of copies wherever possible.

Links are never made for directories or special files.

If several files are given as the source, an existing directory must be specified as the destination.

`–m` (modification) If this option is set, every file copied will have the times of last access and modification set to that of the source.

`–m` not specified:

The modification time will be set to the time of the copy.

`–n` (new) The destination file must be new. If the destination file already exists, the copy command is not executed for this file. For special files this option is assumed. If the source is a directory, this option is meaningless; but it does affect the files in the directory.

`–o` (owner) Only the system administrator is allowed to set this option. If this option is set, every file will have the owner and the group affiliation of the source.

If this option is not set, the person entering the command becomes owner of the copied file.

`–r` (recursive) All subdirectories are copied. This option can be used to copy (recursively) entire file systems.

`–v` (verbose) Logs the copy run on the input device.

source

The *source* can be a file, a directory or a special file. The *source* must exist.

Multiple operands are allowed.

destination

destination is either a file or a directory and must be different from *source*. Note that `copy` does `not` produce an error message if you give the same names for *source* and *destination*.

EXAMPLES

Copy the contents of the directory named `TOOLS` to `TOOLS.BAK`. The directory named `TOOLS.BAK` does not exist, but is created by the `copy` command.

```
$ copy TOOLS TOOLS.BAK
```

SEE ALSO

`cp`(1), `ln`(1).

User's Reference Manual

cp(1)

NAME

`cp` - copy files

SYNOPSIS

Format 1

`cp [ –f ] [ –i ] [ –p ] [ –– ]` *file copyfile*

Format 2

`cp [ –f ] [ –i ] [ –p ] [ –R|–r ] [ –– ]` *file ... directory*

DESCRIPTION

`cp` makes a physical copy (duplicate) of one or more files.

There are two ways to copy with `cp`. The command can copy either:

- one file to another with a different name (**Format 1**), or
- one or more files to a different directory, with the copy retaining the same basename as the corresponding original (**Format 2**).

OPTIONS

Format 1: Copy one file

`cp [–f] [–i] [–p] [––]` *file copyfile*

`–f` (forced) If *copyfile* already exists and a file descriptor cannot be obtained (e.g. no write permission), `cp` deletes the file and recreates it as a copy.

`–i` (interactive) If the named *copyfile* already exists, `cp` will ask you to confirm whether this file may be overwritten. A `y` answer means that the copy should proceed. Any other answer prevents `cp` from overwriting *copyfile*.

If the standard input is not a terminal, this option is ignored and no copying is done.

`–p` `cp` will also duplicate the following characteristics of *file*:

- Modification date and date of last access
- User and user group
- Access permissions

`––` If *file* begins with a dash (–), the end of the command line options must be marked with ––.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

`–e` *extent_op*

Specify how to process extent attribute information. This option specifies the required persistence of extent attributes when processing files which have preallocated space or fixed extent sizes. Valid values for *extent_op* are:

`warn` issue a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified).

`force`
fail the move if extent attribute information cannot be kept.

`ignore`
ignore extent attribute information.

file Name of the original file.

copyfile

Name of the copy file.

If *copyfile* does not yet exist, a new file is created.

Unless the –p option is used, the copy will have the access permissions of the original, modified by the current file-creation mode mask, and the user and group ID of the copy will be those of the user who called cp. But the time of last modification of the copy will be set to the time the copy was made.

Caution:

If a file named *copyfile* already exists and the –i option is not used, the existing file will be overwritten without confirmation, but its mode, owner, and group will be preserved.

If *copyfile* is a link to a file, all links will be retained. The contents of *copyfile* will be overwritten with the contents of *file*.

Format 2: Copy files to another directory

cp [–f] [–i] [–p] [–R|–r] [––] *file ... directory*

–f (f – forced). If *file* already exists in *directory* and a file descriptor cannot be obtained (e.g. no write permission), cp deletes the file and recreates it as a copy.

–i (interactive) If the named *file* already exists, cp will ask you to confirm whether this file may be overwritten. A y answer means that the copy should proceed. Any other answer prevents cp from overwriting *copyfile*.

–p cp will also duplicate the following characteristics of *file*:

- Modification date and date of the last access
- User and user group
- Access permissions

–R|–r

(recursive) If *file* is a directory, cp will copy the directory and all its files, including any subdirectories and their files. Please note that –R or –r are not suitable for copying pipes.

–– If *file* begins with a dash (–), the end of the command line options must be marked with ––.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

–e *extent_op*

Specify how to process extent attribute information. This option specifies the required persistence of extent attributes when processing files which have preallocated space or fixed extent sizes. Valid values for *extent_op* are:

warn issue a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified).

force
fail the move if extent attribute information cannot be kept.

ignore
ignore extent attribute information.

file Name of the original file. You can give a list of names and thus copy several files at once. Each of the copies is assigned the same basic filename (basename) as the corresponding original.

Caution:

If there is a file in *directory* with the same basename as any original, and the –i option has not been set, the existing file will be overwritten without confirmation.

directory

Name of the directory in which the copies are to be placed. This must not be the directory in which the original files are located.

Unless the –p option is set, the copies will have die access permissions of the original, modified by the current file-creation mode mask, and the user and group ID of the copies will be those of the user who

called `cp`. The time of last modification of each copy will be set to the time the copy was made.

ERROR MESSAGES

`cp: cannot access` *file*

The named *file* does not exist.

`cp: cannot open` *file*

You have no read permission for *file*.

`cp: cannot create` *file*

You do not have write permission for the directory in which *file* is to be created, or the named directory does not exist.

`cp: <`*dir*`> directory`

dir is a directory and cannot be copied (**Format 1**), or you have not set the `–R`|`–r` option (**Format 2**).

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

Answers to yes/no queries must be given in the language appropriate to the current locale.

EXAMPLES

Example 1

The file `techlit` is to be copied before it is changed. The copy is to be called `tl` and to be located in the same directory as `techlit`.

```
$ cp techlit tl
```

Example 2

All files from the current directory with names beginning with `file` are to be copied into the directory `/home/do/save`, and their times of last modification are to be duplicated.

```
$ cp –p fil* /home/do/save
$ ls –l /home/do/save
total 4
-rw------  1  dober  gr1        37 Nov 11 11:11  file1
-rw------  1  dober  gr1        97 Apr 01 13:24  file2
-rw------  1  dober  gr1       116 Dec 31 12:13  file3
-rw------  1  dober  gr1       381 Feb 16 08:08  file4
```

SEE ALSO

`chmod(1)`, `cpio(1)`, `ln(1)`, `mv(1)`, `pax(1)`, `rm(1)`.

User's Reference Manual

cpio(1)

NAME

`cpio -` copy files and directories in and out

SYNOPSIS

Format 1

`cpio –o`[*add_option ...*] [*single_option ...*]

Format 2

`cpio –i`[*add_option ...*] [*single_option ...*] [*pattern ...*]

Format 3

`cpio –p`[*add_option ...*] [*single_option*] *directory*

DESCRIPTION

The `cpio` command has three functions:

- it copies one or more files into an archive file (**Format 1**)
- it retrieves files from an archive previously created with `cpio`, `tar` or `pax` (**Format 2**)
- it copies files into a directory (**Format 3**)

Only the main options `–o`, `–i`, `–p`, and `–e` are detailed in the descriptions of the above three formats. The remaining options are explained in the section on **ADDITIONAL and SINGLE OPTIONS**. The various options permitted with each format are indicated in the format descriptions.

OPTIONS

Format 1: Copy files out

`cpio –o`[*add_option ...*] [*single_option ...*]

add_option

`a`, `A`, `B`, `c`, `L`, `q`, `v`, `V`

single_option

`–C` *bufsize*, `–H` *header*, `–O` *archive* [`–M` *message*], `–R` *userid*

Main options

`–o` (out) `cpio` reads a list of path names of plain files from standard input and copies these files along with some status information to standard output in a special archive format. Output is padded to a 512-byte boundary. The number of 512-byte blocks copied is written to standard error.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

`–e` *extent_op*

Specify how to process extent attribute information. This option specifies the required persistence of extent attributes when processing files which have preallocated space or fixed extent sizes. Valid values for *extent_op* are:

`warn` issue a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified).

`force`
fail the file save if extent attribute information cannot be kept.

`ignore`
ignore extent attribute information.

Format 2: Copy files in

`cpio –i`[*add_option ...*] [*single_option ...*] [*pattern ...*]

add_option

b, B, c, d, f, k, m, r, s, S, t, u, v, V, 6

single_option

−C *bufsize*, −E *file*, −H *header*, −I *archive* [−M *message*], −−

Main options

−i (in) cpio reads from an archive created beforehand with cpio/tar/pax. The archive must be sent to cpio via standard input. cpio fetches from input those files whose names match with *pattern*. These files are created and copied into the current directory tree in accordance with the additional options that you specify. The extracted files have the same permissions as the files copied out with cpio/tar/pax, while the user ID and group ID are those of the user calling cpio −i. It is only when the system administrator calls cpio −i that the extracted files retain the same user or group ID as the files copied out with cpio/tar/pax. Only the system administrator can extract special files.

The number of 512-byte blocks read is output by cpio to standard error.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

−e *extent_op*

Specify how to process extent attribute information. This option specifies the required persistence of extent attributes when processing files which have preallocated space or fixed extent sizes. Valid values for *extent_op* are:

warn issue a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified).

force
fail the file restore if extent attribute information cannot be kept.

ignore
ignore extent attribute information.

pattern

pattern specifies the files to be selected from the input. All shell metacharacters for file name generation can be used in defining *pattern*. The maximum length for a *pattern* is 1024 characters.

pattern not specified:

Has the same effect as entering an asterisk *, i.e. all files are extracted, even those with names that begin with a dot.

Format 3: Copy files to a directory

cpio −p[*add_option* ...] [*single_option*] *directory*

add_option

a, d, l, L, m, u, v, V

single_option

−R *userid*, −−

Main options

−p (pass) cpio reads a list of path names of plain files, creates the corresponding files, and copies them to the named *directory*, in accordance with the additional options that you specify. The number of 512-byte blocks copied by cpio is written to standard error.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

−e *extent_op*

Specify how to process extent attribute information. This option specifies the required persistence of extent attributes when processing files which have preallocated space or fixed extent sizes. Valid values for *extent_op* are:

warn issue a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified).

`force`
fail the file save or restore if extent attribute information cannot be kept.

`ignore`
ignore extent attribute information.

directory
Name of the destination directory to which the files are to be copied. The path name must not be more 1024 characters long.

ADDITIONAL and SINGLE OPTIONS (for all 3 formats)

The additional options must follow the main options immediately and be specified without a minus sign or blanks, e.g. `cpio -pdl`.

Single options are specified after the additional options; they must be preceded by a minus sign and followed by associated arguments, e.g. `cpio -oBcv -O /dev/ios0/stape004`.

`a` (access time) Resets access times of input files after they have been copied.

Access times are not reset for linked files when option `l` is also specified.

Options `a` and `m` are mutually exclusive and must not be combined.

`A` Appends files to an existing archive. The `-A` option can only be combined with single option `-O` *archive*. It is only valid for archives which are files or which are on floppy disk(s) or hard disk partitions. Open-reel tapes and tape cartridges are not supported.

`b` (bytes) Reverses the byte sequence within each word. A word is equivalent to 4 bytes.

This option can only be combined with main option `-i`.

`B` (block) Input/output is to be blocked 5120 bytes to the record.

The default buffer size is 512 bytes when this option and single option `-C` are not used. When listing the blocks which it is reading in or out, `cpio` generally assumes a block size of 512 bytes. Option `-B` is only meaningful with data directed to or from a character special file.

Archives written with a particular block size may not be read with a smaller block size. In contrast, smaller block sizes are detected automatically when they are read from the archive.

This option must not be combined with main option `-p`.

`c` (compatible) `cpio` writes or reads header information in US ASCII format for compatibility reasons.

You should always use this additional option (or single option `-H`) when the source and target machines are of different types.

Options `c`, `6`, and `-H` are mutually exclusive and must not be combined.

Note:

Archives created with `c` in Version 5.40 cannot be read on systems that have an earlier UNIX version installed. Use single option `-H odc` if you want your archive to be readable on a system with an earlier operating system version. A header created with `-H odc` corresponds to a header created with `c` on an earlier version.

To ensure compatibility with System V Release 3.2 and earlier releases, `cpio` does not usually back up files with inode numbers above 65535. To ensure that such files are backed up, you should use additional option `c`.

Attention: You must use this format for files larger than 2 GB. A warning is issued in this case.

`-C` *bufsize*
Input/output is to be blocked *bufsize* bytes to the record, where *bufsize* must be a positive integer. For floppy disks, *bufsize* must be a multiple of 512 bytes (one floppy sector). The default buffer size is 512 bytes when this option and additional option `B` are not used.

This option must not be combined with main option `-p`.

`d` (directory) Directories are automatically created as needed.

–E *file*

(extract) Specify an input file *file* that contains a list of file names to be extracted from the archive. Only one file name per line may be given; however, you can additionally specify a pattern as well.

f Only the files not matching *pattern* are copied.

–H *header*

cpio reads or writes header information in *header* format. You should always use this option or the c option when the source and target machines are of different types.

The following *header* formats are permitted:

crc ASCII header with expanded inode numbers and an additional per-file checksum

ustar
IEEE/P1003 Data Interchange Standard header and format

tar tar-style header and format

odc ASCII header with small inode numbers

When exchanging data with a computer that has a SINIX version earlier than SINIX V5.40 installed, you should use single option –H odc. Headers created with –H odc in SINIX V5.40 are equivalent to headers created with c on earlier SINIX versions.

Path names are restricted to 256 characters for the binary (the default) and –H odc header formats. Otherwise, a maximum of 1024 characters are permitted.

Inode numbers of files backed up with –H odc must not be above 262143 (777777 octal).

–I *archive*

(input) This option can only be combined with main option –i.

cpio reads input from the named *archive*.

k This option can only be combined with main option –i.

cpio attempts to skip any corrupted file headers and I/O errors that it encounters. If you want to copy files from a medium that is partially corrupted, this option lets you read only those files with good headers. In the case of archives that contain other cpio archives, cpio may terminate prematurely if an error occurs. cpio will then find the next good header, which may be one for a smaller archive, and terminate when it encounters the smaller archive's trailer.

l (link) This option can only be combined with option –p.

To save storage space when working with a large number of files, whenever possible, the files should be linked rather than copied to another directory.

L (symbolic links) If this option is set, cpio follows symbolic links. The result of copying in a symbolic link on an earlier version will be a regular file that contains the path name of the file referenced by the symbolic link. Note that if you use the L option with cpio in conjunction with find(1), then you must use find's follow option.

m (modification time) Retains the previous file modification time. This option is not effective on directories. Options m and a are mutually exclusive and must not be combined.

–M *message*

(message) Defines a message to be used when switching media. When you use the –O or –I options and specify a character special file, you can use this option to define the message that is printed when you reach the end of the volume. One %d can be placed in *message* to print the sequence number of the next volume needed to continue. You may specify more than one word for *message*, provided you enclose the string in double quotes ("...").

–O *archive*

(output) This option can only be combined with main option –o.

cpio directs its output to the named *archive*.

–q (quick) cpio in fast mode. The ipc facilities are allocated now. A child is forked to do the reading and the parent will do the writing to the device.

The –q option can only be combined with main option –o.

r (rename) Allows files to be renamed interactively during copying. cpio prompts for a new name for each file. If you answer by simply pressing <RETURN>, the file will not be renamed.

The r option does not work with open-reel tapes, tape cartridges or Video 8 cartridges.

–R *userid*
(reassign) This option is only available to the system administrator. cpio reassigns ownership and group information for each file in accordance with the specified *userid*, which must be a valid user ID from /etc/passwd file.

s (swap) This option can only be combined with main option –i. It causes bytes to be swapped within each half word. A word is equivalent to 4 bytes.

S (swap) This option can only be combined with main option –i. It causes halfwords to be swapped within each word. A word is equivalent to 4 bytes.

t (table of contents) Produces a table of contents of the source from which the files are to be read. No files are copied.

u (unconditional) Replaces an existing file with an extracted file with the same path name even if the existing file has a more recent access date than the one being extracted.

u not specified:

Existing files are replaced by extracted files (with the same path name) only if the existing files are older than the extracted ones.

v (verbose) Causes the names of files to be output as they are being copied in or out. If v is used in conjunction with t, detailed information about the files will be provided as well.

V (special verbose) Prints a dot for each file read or written. This option is useful to assure the user that cpio is working without providing detailed information.

6 This option may only be used in combination with option –i.

cpio processes a UNIX System Sixth Edition archive format file. Options c, –H and 6 are mutually exclusive.

–– If the first argument begins with a dash (–), the end of the command-line options must be marked with ––.

MESSAGES

```
Warning: just archiving file filename > 2GB.
cpio format unable to handle filename - filesize > 2GB.
Please use -c Option.
Error: Old format cannot support expanded types on filename
```

You did not use the c option. To ensure compatibility, the way in which cpio handles the inode of its files means that inodes beyond 64 Kb cannot be processed unless you use the c option.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. In regular bracketed expressions, the LC_COLLATE environment variable governs the scope of ranges, equivalence classes and collating elements, while the LC_CTYPE environment variable governs the scope of character classes.

The LC_TIME environment variable governs the format of date and time strings in archive table-of-contents listings produced by the –v option.

If LC_MESSAGES, LC_COLLATE, LC_CTYPE or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other

environment variables that affect internationalization.

EXAMPLES

Example 1

All files in the current directory with names that end in `.f` are to be copied out to an archive file named `f.progs`:

```
$ find . –name '*.f' –print | cpio –o >f.progs
56 blocks
```

`find . –name '*.f'` searches the current directory for file names that end in `.f` and writes them to the standard output, which is piped (|) to the standard input of `cpio`. `cpio –o` then reads the file names and copies the appropriate files along with status information to standard output. The "greater than" symbol > redirects the standard output to the file `f.progs`. When it has finished, `cpio` writes the number of copied blocks on standard error.

Example 2

The contents of a directory and all its subdirectories are to be copied to a tape cartridge:

```
$ find . –print | cpio –oBv –H odc –O /dev/ios0/stape004
```

`find` searches the current directory and any subdirectories and writes the relative path names of all the files that it finds to standard output. The standard output of `find` is piped (|) to the standard input of `cpio`. `cpio` reads the file names and copies the files to the tape cartridge, retaining the structure of the directory tree. The names of the files being copied are echoed on the standard output (option `v`). Option `B` is used to increase the buffer size, option `–H odc` to ensure that the tape is readable on machines of different types. You should use the `–H odc` option whenever possible to ensure compatibility with other systems.

Example 3

All files whose names end with `.c` (= pattern) must be extracted from the `file.cpio` archive:

```
$ cpio –ivmdB *.c <file.cpio
```

Example 4

An archive on a magnetic tape cassette on device `/dev/ios0/rstape003` is to be extracted:

```
$ cpio –ivdB /dev/ios0/rstape003
```

Example 5

The directory named `dir.old` is to be copied together with all its subdirectories. The duplicate directory is to be named `dir.new`:

```
$ cd dir.old
$ find . –depth –print | cpio –pdl dir.new
```

The `cd` command switches you to the directory `dir.old`. `find` then searches this directory and any subdirectories and writes the names of all the files that it finds to standard output. The standard output of `find` is piped (|) to the standard input of `cpio`. `cpio` reads the file names and copies the files to the directory `dir.new` (`–p dir.new`). Directories are created as needed (`–d`), i.e. all subdirectories of `dir.old` are copied recursively. Since option `–l` is also specified, the files, where possible, are not actually copied but linked.

SEE ALSO

ar(1), cat(1), echo(1), find(1), ls(1), mt(1), pax(1), tar(1), ar(4), archives(4).

User's Reference Manual

crontab(1)

NAME

`crontab` - regularly execute commands at specific times

SYNOPSIS

Format 1
`crontab` [––] [*file*]

Format 2
`crontab –l` [*login_name*]

Format 3
`crontab –r` [*login_name*]

Format 4
`crontab –e` [*login_name*]

DESCRIPTION

`crontab` is used to have commands, shell scripts or executable programs executed regularly at specific times.

You can use `crontab` to:

- schedule command jobs (**Format 1**)
- have scheduled jobs listed (**Format 2**)
- delete scheduled jobs (**Format 3**)
- edit scheduled jobs (**Format 4**)

Before the call

If the file `/etc/cron.d/cron.allow` exists, you can only use `crontab` if your login name appears in it. If the file `/etc/cron.d/cron.allow` does not exist, you can only use `crontab` if your login name does *not* appear in the file `/etc/cron.d/cron.deny`. If neither `/etc/cron.d/cron.allow` nor `/etc/cron.d/cron.deny` exists, only the system administrator is allowed to use `crontab`.

If only an empty deny file exists, for example, everyone is allowed to use `crontab`.

Only the system administrator is allowed to create and modify the allow and deny files. Each line in these files contains precisely one login name.

If a privileged user modifies another user's `crontab` file, resulting behavior may be unpredictable. Instead, the privileged user should first use `su` to switch to the other user's login name and thus obtain the permissions needed to make effective changes to that user's `crontab` file.

Caution:

If you inadvertently enter the `crontab` command with no arguments, do *not* attempt to get out with CTRL-D. This will cause all entries in your `crontab` file to be removed. Instead, use DEL to exit.

OPTIONS

Format 1: Scheduling command jobs

`crontab` [––] [*file*]

To schedule a command job with `crontab` you have to specify:

- the command (or shell script or program) that is to be executed, and
- the times at which it is to be executed (e.g. every Friday or every January 15).

When you call `crontab` in the above format, `crontab` copies these specifications into your `crontab` file

/var/spool/cron/crontabs/*login_name*. The /etc/cron process checks every minute whether there are commands in the crontab file which need to be executed, and if there are any, it has them executed.

To execute the specified *command*, crontab invokes a new shell (sh) from your home directory. crontab supplies a default environment for every shell, defining HOME, LOGNAME, SHELL (/bin/sh), and PATH (:/bin:/usr/bin:/usr/lbin). If you wish to have your .profile executed, you must specify this explicitly in the crontab file.

If neither the standard output nor the standard error output have been redirected for the commands in the crontab file, both the standard output and the standard error output will be sent to you by mail.

There can be at most one crontab file per login name. If the crontab file /var/spool/cron/crontabs/*login_name* does not exist, it is created when crontab is called. If it already exists, its contents are overwritten. In other words, if you wish to schedule additional command jobs with crontab, you should use the –e option.

–– If *file* begins with a dash (–), the end of the command-line options must be marked with ––.

file Name of the file containing the commands and the times at which they are to be executed.

file must be in the format described below (see **STRUCTURE OF A crontab FILE**).

file not specified:

crontab reads the text entered from the standard input. The text that you enter in this case must be in the format described below (see **STRUCTURE OF A crontab FILE**).

Format 2: Listing scheduled jobs

crontab –l [*login_name*]

–l (list) crontab lists the contents of your crontab file /var/spool/cron/crontabs/*login_name*.

If the file is not available, an error message is issued (see **ERROR MESSAGES**).

login_name

If a *login_name* is specified, the crontab file of the corresponding user is listed. However, only the system administrator is permitted to specify a *login_name*.

Format 3: Deleting scheduled jobs

crontab –r [*login_name*]

–r (remove) crontab deletes your crontab file /var/spool/cron/crontabs/*login_name*.

login_name

If a *login_name* is specified, the crontab file of the corresponding user is deleted. However, only the system administrator is permitted to specify a *login_name*.

Format 4: Editing scheduled jobs

crontab –e [*login_name*]

–e (edit) The –e option allows you to edit a copy of your crontab file or to create an empty file to edit (if crontab does not exist) and save this file as the current crontab file. The VISUAL environment variable determines which editor is called. If VISUAL is not defined, the EDITOR environment variable is checked, and if EDITOR is not defined, ed is invoked.

login_name

If a *login_name* is specified, the crontab file of the corresponding user is edited. However, only the system administrator is permitted to specify a *login_name*.

STRUCTURE OF A crontab FILE

Your crontab file, or the text that you enter from standard input, (see **Format 1**) must be in the following format:

Text comprising:

- lines to define a command job, and

- comment lines, if required.

Blank lines are not permitted.

Command job definition lines

A line defining a command job consists of six fields separated by blanks or tabs. The fields contain the following information:

1	2	3	4	5
Minute	*Hour*	*Day of month*	*Month*	*Day of week*

Fields 1 to 5 specify the times at which the command in field 6 is to be executed.

Minute
: Range of values: 0-59 or *pattern*

Hour Range of values: 0-23 or *pattern*

Day of month
: Range of values: 1-31 or *pattern*

Month
: Range of values: 1-12 or *pattern*

Day of week
: Range of values: 0-6 (0 = Sunday) or *pattern*

Command
: Name of the command that is to be executed at the specified time.

  If your command needs to read data from standard input, you have to use the percent character `%`. Each `%` character in the command field that has not been escaped with a backslash `\` is interpreted as a newline character. The shell executes the content of the command field only up to the first unescaped percent or newline character; the remainder of the field is passed to the command as standard input (see *Example 1*).

pattern may be:

- an asterisk `*` (standing for all legal values)
- a range described as *number-number*
- a comma-separated list of legal numbers or ranges, e.g.: 1,3,5 or 1–3,5

Specifying the day

Two fields are available for specifying the day; field 3 (day of month) and field 5 (day of week).

If you enter a number, a range or a list in the third field as well in the the fifth field, both specifications are valid, independently of each other.

For example, if your `crontab` file contains the line:

`0 0 1,15 3 1` *command*

then *command* will be executed on March 1 and March 15 as well as on every Monday in March.

If you wish to specify the day in one field only, you must enter an asterisk in the second field. Thus if your `crontab` file contains the line:

`0 0 * 3 1` *command*

then *command* will be executed on every Monday in March.

Comment lines

A comment line has the hash character # in column one and may be followed by any arbitrary text. You can use comment lines in the `crontab` file to explain the function of the command job or to arrange the job into sections for the sake of clarity. Please note that the input text that you pass to `crontab` can contain any number of comment lines but must not include any blank lines.

ERROR MESSAGES

`crontab: you are not authorized to use cron. Sorry.`

See `Before the call` above.

`Format 1`

`crontab: can't open your crontab file.`

You have called `crontab` with the name of a file which does not exist or cannot be accessed.

`crontab: error in previous line; ...`

If any line of the input text that you pass to `crontab` is not in the correct format, `crontab` displays the relevant line followed by the above error message. The ellipses (...) shown here are replaced by a more detailed description of the error, e.g.:

`unexpected character found in line`

An illegal character was found in the line.

`number out of bounds`

A specified number is not in the permitted range.

`Format 2`

`crontab: can't open your crontab file.`

Your `crontab` file `/var/spool/cron/crontabs/`*login_name* does not exist and therefore cannot be read with `crontab -l`.

ENVIRONMENT VARIABLES

`VISUAL`
: The `VISUAL` environment variable determines which editor is used when the `-e` option is specified.

`EDITOR`
: If `VISUAL` is not set, the `EDITOR` environment variable determines which editor is used when the `-e` option is specified. The default value is `vi`.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

You want the system to remind you on May 15 every year that it is Aunt Jemima's birthday. To do this, you set up a file called `birthday` with the following contents:

```
0 0 15 5 * echo Today is Aunt Jemima's birthday !!!
```

Please note that since you only wish to make an entry for the day of the month and not for the day of the week, you must enter an asterisk in the "day of week" field.

You now pass the file named `birthday` to your `crontab` file:

```
$ crontab birthday
```

The system will then send you the appropriate message via `mail` at 00.00 hours on May 15 of each year. This will obviously only function if the system has not been turned off during that night.

Example 2

Let us assume that on every Monday, at 1:30 p.m., you want to remind yourself and a co-worker of your aerobics class that evening. Your reminder is to be displayed on the screen. You are at terminal tty007 and she is at terminal tty014. You can also include comment lines in the input text so that your text is easier to read with `crontab -l` at a later stage. Your `crontab` file could then be set up as follows:

```
# *   = all legal values
# n-n = range
# n,n = list
# Day of week: 0 = Sunday
#
# Min  Hrs  Day(Mon)  Mon  Day(Wk)  Command
#
30   13      *        *    1      echo Aerobics this evening ! >/dev/tty007
30   13      *        *    1      echo Aerobics this evening ! >/dev/tty014
```

Every Monday at 1:30 p.m., you and your colleague will now receive the message "Aerobics this evening !" at your respective terminals (and not in your mailboxes).

Example 3

Every Monday, you want to `mail` the message below to user `melody`, who works on a system called `jukebox`:

```
It may be blue
But it will pass too!
```

You do this by putting the following line in your `crontab` file:

```
0 0 * * 1 mail melody@jukebox %It may be blue %But it will pass too!
```

FILES

`/etc/cron.d/cron.allow`
: List of login names with permission to use `crontab`. One login name is entered per line.

`/etc/cron.d/cron.deny`
: List of login names explicitly denied permission to use `crontab`. One login name is entered per line.

`/var/spool/cron/crontabs/`*login_name*
: `crontab` file of the user identified by *login_name*.

SEE ALSO

`at`(1), `atq`(1), `atrm`(1), `mail`(1), `sh`(1), `su`(1), `vi`(1), `cron`(1M), `queuedefs`(4).

User's Reference Manual

crypt(1)

NAME

`crypt` - encode/decode text

SYNOPSIS

Format 1

`crypt` [`–c`] [*key*]

Format 2

`crypt` [`–k`]

DESCRIPTION

`crypt` encodes and decodes the contents of files. `crypt` reads from standard input and writes to standard output. Files encrypted without the `–c` option are compatible with those created or edited by the editors `ed`, `edit`, `ex` and `vi` in encryption mode.

OPTIONS

Format 1:

`–c` (compatible) If the `–c` option is set, then

- files encrypted on systems in the MIPS range can be decrypted on systems in the Intel range
- files encrypted on systems in the Intel range can be decrypted on systems in the MIPS range

key You may enter any string (1 – 8 characters) of your choice as the key with which you define the encryption or decryption mechanism.

key not specified:

You enter the required encryption key at the terminal. Your input will not be echoed to the screen.

Format 2:

`–k` `crypt` uses the value of the `CrYpTkEy` variable as the encryption or decryption key (see **ENVIRONMENT VARIABLE**). The case of the letters in the variable name must be exactly as shown here.

Security criteria

The security of encrypted files essentially depends on the following factors:

- The underlying encryption method must not be detectable.
- It must be impossible to directly search for the sequence of keystrokes entered for encryption.
- If you concatenate two or more files separately encrypted with the same key into a single file and then try to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

ENVIRONMENT VARIABLE

`CrYpTkEy`

The `CrYpTkEy` variable defines the encryption/decryption key for `crypt`. The case of the letters in the variable name must be exactly as shown here.

EXAMPLES

Example 1

The file `Grandma.ltr` contains the following message:

```
Dearest Granny,
I am sending you a secret message today so that the big bad wolf
won't be able to find out when I'll be visiting you again in the
```

```
forest with wine, cakes, and other goodies.
Love, Little Red Riding Hood.
```

It is now vital that this letter be encrypted! To do this, Little Red Riding Hood uses the key `abcd`.

Encryption and redirection to `topsecret-file`:

```
$ crypt abcd < Grandma.ltr > topsecret-file
```

`topsecret-file` now contains unreadable garbage made up of special characters and meaningless character sequences.

Grandmother can now decrypt the secret file, but only because Red Riding Hood first asked the good hunter to give her the keyword, and he has told her that it is `abcd`:

Decryption and output on the screen:

```
$ crypt < topsecret-file
Enter key: abcd
Dearest Granny,
I am sending you a secret message today so that the big bad wolf
won't be able to find out when I'll be visiting you again in the
forest with wine, cakes, and other goodies.
Love, Little Red Riding Hood.
```

If Little Red Riding Hood encrypted her message on a system in the MIPS range and Grandmother has an Intel-based machine (!), Grandmother needs to enter:

```
$ crypt –c < topsecret-file
```

Example 2

The contents of `letter` are encrypted, and the encrypted output is stored in `letter.secr`. `crypt` is called with the `–k` option. The keyword defined with the `CrYpTkEy` variable is the string `flatnix`.

```
$ CrYpTkEy=flatnix; export CrYpTkEy □
$ crypt –k < letter > letter.secr
```

SEE ALSO

`ed(1)`, `edit(1)`, `ex(1)`, `makekey(1)`, `pg(1)`, `ps(1)`, `stty(1)`, `vi(1)`.

csh(1)

NAME

`csh` - C shell

SYNOPSIS

`csh` [*option* ...] [*file* [*argument* ...]]

DESCRIPTION

`csh`, the C shell, is a command interpreter like the standard (Bourne) shell `sh`, and like `sh` it also provides the essential elements of a programming language. The syntax of this programming language is reminiscent of the C language (hence the name `csh`).

Like the standard Bourne shell, the C shell also supports I/O redirection and variables, command substitution, file name generation (substitution), and so on. There are, however, more options for interactive use than in the Bourne shell. These include:

- completion of file names and login names,
- history substitution,
- command aliasing,
- job control

In addition, the C shell provides a number of built-in commands that are not available in the Bourne shell.

OPTIONS

`-b` (break) Subsequent command-line arguments are not interpreted as C shell options. This allows you to pass options to a shell script without them being interpreted as options for `csh`.

`-c` When you use this option, you must specify the name of a shell script for *file*. All *arguments* that follow *file* are interpreted as arguments for *file* and are placed in `argv`, the argument-list variable. The script is executed in a subshell.

`-e` (exit) Causes `csh` to exit immediately if a command terminates abnormally or returns a non-zero exit status.

`-f` (fast start) Tells `csh` not to read the `.cshrc` file or the `.login` file upon startup.

`-f` not specified:

The `.cshrc` file is read at startup. If `csh` is started as a login shell, the `.login` file is also read.

`-i` (interactive) Forces `csh` to prompt for command-line input, even if the standard input does not appear to be a terminal.

`-n` (no execution) `csh` parses (interprets), but does not execute commands. This option can be used to check C shell scripts for syntax errors.

`-s` (standard input) `csh` reads commands from standard input.

`-t` Tells `csh` to read and execute a single command line. A \ (backslash) can be used to escape newlines so as to continue the command line onto subsequent input lines.

`-v` (verbose) Sets the predefined variable `verbose`, which causes command input to be echoed on standard error after history substitution (but before other substitutions) and before command execution.

`-V` Tells `csh` to set the `verbose` variable before reading the `.cshrc` file.

`-x` Sets the predefined variable `echo`, which causes command input to be echoed on standard output after all substitutions and just before command execution.

`-X` Tells `csh` to set the `echo` variable before reading the `.cshrc` file.

file [*argument* ...]

file [*argument* ...] must not be combined with the `–i`, `–s` or `–t` option. *file* is taken to be the name of a command or shell script to be executed. It is passed as argument zero.

argument

Argument for the named command or script *file*. You may specify more than one argument. The arguments are placed in the argument-list variable `argv`.

file [*argument* ...] not specified:

`csh` reads commands from standard input.

FUNCTIONALITY AND USAGE OF THE C SHELL

This section is divided into the following subsections:

- Starting and terminating the C shell
  - the files `.cshrc`, `.login` and `.logout`
- Interactive operation
  - Cshell prompts
  - how the C shell processes the command line
- Non-interactive operation
- File name/login name completion
- Lexical analysis
  - splitting the command line into words
- Command line parsing
  - simple commands
  - pipelines
  - linking pipelines
  - grouping pipelines within parentheses
  - commands or pipelines that run in the background
- History substitution
- Command aliasing
- I/O redirection
- Variable substitution
  - creating, displaying and deleting variables
  - variables used by the shell
  - referencing the value of a variable
  - escape characters and variable substitution
  - timing of variable substitution
  - shell parameters
- Command substitution
- File name substitution
- Expressions and operators
  - grouping
  - arithmetic operators
  - logical operators
  - relational operators
  - file attribute inquiries

  - inquiry as to the success of a command
- Control flow
  - loops and branches
- Command execution
  - how the shell searches for a command
  - how commands are executed
- Signal handling
- Job control
  - jobs and job numbers
  - changing the status of jobs
  - referencing a specific job
  - I/O for background jobs
  - status reporting by the C shell
- Cshell built-ins
  - the built-in commands `:`, `alias`, `bg`, `cd`, `chdir`, `dirs`, `echo`, `eval`, `exec`, `exit`, `fg`, `glob`, `hashstat`, `history`, `jobs`, `kill`, `limit`, `login`, `logout`, `nice`, `nohup`, `notify`, `onintr`, `popd`, `pushd`, `rehash`, `repeat`, `set`, `setenv`, `shift`, `source`, `stop`, `suspend`, `time`, `umask`, `unalias`, `unhash`, `unlimit`, `unset`, `unsetenv`, `wait`, `%` and `@`.
  - control-flow statements
- Environment variables and predefined shell variables
- Cshell scripts
  - the nature of C shell scripts
  - starting C shell scripts
  - comments
  - which shell runs the script

Starting and terminating the C shell

When first started, the C shell normally performs commands from the `.cshrc` file in your home directory (see the `–f` option), provided that it is readable and you either own it or your real group ID matches its group ID.

If the shell is invoked with a name that starts with `–` (as when started by `login`), the shell runs as a login shell. In this case, after executing commands from the `.cshrc` file, the shell executes commands from the `.login` file in your home directory. The `.login` file is subject to the same permission requirements as `.cshrc`. Typically, the `.login` file contains commands specifying the terminal type and the environment.

As a login C shell terminates, it performs commands from the `.logout` file in your home directory; the `.login` file is subject to the same permission requirements as `.cshrc`.

Interactive operation

After startup processing is complete, an interactive C shell displays the prompt *hostname*`%` (for ordinary users) or *hostname*`#` (for the system administrator). If you wish, you can change the prompt by redefining the `prompt` variable.

After displaying the prompt, the shell reads the commands entered at the terminal and then processes each input line as follows:

1. It reads a line of command input and splits it into words (see **Lexical analysis**).
2. It places this sequence of words on the history list (see **History substitution**).

3. It then parses the sequence of words (see **Command line parsing**).
4. It performs alias substitution (see **Command aliasing**).
5. It expands history references (see **History substitution**).
6. It substitutes values for variables (see **Variable substitution**).
7. It substitutes file names for patterns (see **File name substitution**).
8. It redirects I/O as required (see **I/O redirection**).
9. It replaces commands enclosed within backquotes `...` with their output (see **Command substitution**).
10. It executes the commands in the input line (see **Command execution**).

Non-interactive operation

When running non-interactively, the C shell does not display a prompt. A non-interactive C shell can execute a command supplied as an argument on its command line, or interpret commands from a script.

File name/login name completion

If the `filec` variable is set accordingly, an interactive C shell can complete a partially typed file name or login name.

- If an unambiguous partial file name is followed by an ESC character, the shell fills in the missing characters of a matching file name from the working directory.
- If a partial file name is followed by the EOF character <CTRL-D>, the shell lists all matching file names in the current directory. It then prompts once again, supplying the incomplete command line typed in so far. You can then enter one of the listed file names.
- If the last (partial) word begins with a tilde (`~`) and is followed by an ESC character, the shell attempts to complete this word with an existing login name, rather than a file in the working directory (see **File name substitution**).

  If a partially entered word preceded by a tilde is followed by the EOF character <CTRL-D>, the shell lists all matching login names. It then prompts once again, supplying the incomplete command line typed in so far. You can then enter one of the listed login names.

The terminal bell rings to signal errors or multiple matches; you can suppress this by setting the `nobeep` variable.

You can exclude file names with certain suffixes by listing these suffixes in the `fignore` variable. If, however, the only possible completion for a file name includes a suffix in the list, the file name in question is not ignored. `fignore` does not affect the listing of file names by the EOF character.

See *Examples 1* and *2* at the end of the C shell description.

Lexical analysis

The C shell splits input lines into words according to the following rules:

- Space and tab characters normally serve as word separators.
- The special characters &, |, ;, <, >, (, and) form separate words; if paired (e.g. || or >> or |&), the pairs form single words. These shell metacharacters can be part of other words if their special meaning is suppressed by preceding them with a backslash (e.g. a\|b is one word; a|b forms three words).
- A newline preceded by a backslash is equivalent to a space character.
- A string enclosed in matched pairs of single quotes ('...'), double quotes ("..."), or backquotes (`...`) forms a partial word; metacharacters in such a string (including any space or tab characters) do not form separate words. Within pairs of single quotes, double quotes, or backquotes, a newline preceded by a backslash is interpreted as a newline character (not a space).
- When the shell's input is not a terminal, the hash character # introduces a comment that continues to the end of the input line. All characters that form part of the comment are ignored by the shell (see **C shell scripts**). The special meaning of the # character is suppressed when it is preceded by a backslash or

enclosed within matching single or double quotes.

- The following two commands

  ```
  who>/tmp/whos_on
  who>>/tmp/whos_on
  ```

  comprise three words each: `who`, `>` or `>>`, and `/tmp/whos_on`. However, the command

  ```
  "who am I"
  ```

  is taken to be one word. The C shell interprets the word `who am I` as the name of a program to be executed.

Command line parsing

The command line for the C shell is structured as follows:

- A simple command is composed of a sequence of words. The first word (that is not part of an I/O redirection) identifies the command to be executed. All other words with no special meaning for the C shell are passed as arguments to the command to be executed.
- A simple command, or a set of simple commands separated by | or |& characters, forms a pipeline. The possible combinations and their meanings are:

  command1 | *command2*
  : The standard output of *command1* is redirected to the standard input of *command2*.

  command1 | &*command2*
  : Both the standard output and the standard error of *command1* are redirected to the standard input of *command2*.

- Two or more pipelines can be connected into a sequence by separating them with a colon or the character pair && or || as shown below:

  pipeline1 ; *pipeline2*
  : The two pipelines are executed sequentially.

  pipeline1&&*pipeline2*
  : Conditional sequence in which *pipeline2* is executed only if *pipeline1* is successful.

  pipeline1 | | *pipeline2*
  : Conditional sequence in which *pipeline2* is executed only if *pipeline1* is not successful.

- A pipeline or sequence of pipelines can be enclosed within parentheses (...) to form a simple command, which in turn may be a component in a pipeline or a sequence of piplines.
- A sequence of pipelines will be executed asynchronously (in the background) if an ampersand character & is appended to it; in this case, instead of waiting for the sequence to finish before issuing a prompt, the shell displays the job number and associated process IDs, and prompts immediately (see **Job control**).

History substitution

History substitution allows you to repeat the execution of previously entered commands or to use words from previous command lines in your current command line. This simplifies spelling corrections and the repetition of complicated commands.

Command lines are saved in the history list, the size of which is governed by the value of the `history` variable. The most recent command is always saved in the history list.

Each command line entry that is saved in the history list is known as an `event` and is assigned an event number in ascending order.

History references may be entered anywhere on the command line, but they cannot be nested.

Each history reference relates to a specific event (see *Event designators*). To expand a history reference, the shell searches the history list backwards, i.e. from the last event to the first. If it finds the required event, it selects the designated words (see *Word designators*) and changes them as required (see *Modifiers*). The string generated in this way is then inserted in place of the history reference on the current command line. The

resulting new command line is then placed on the history list and also displayed on the screen. Finally, the shell performs further substitutions such as variable substitutions, file name substitutions, etc. and executes the command. See *Examples 1* and *7* at the end of the C shell description.

History references have the following format:

`!`[*event-designator*][`:`*word-designator*][`:`*modifier* ...]

`!` Starts a history substitution, except when followed by a space character, a tab, a newline, an equal sign or a left parenthesis.

You can use the `histchars` variable to define some other character instead of `!` to begin a history substitution. The character can be escaped with a backslash to suppress its special meaning.

In special cases history references may also begin with a caret `^` (see *Quick substitution*).

Event designators

An event designator is a reference to a specific command-line entry (event) in the history list. The following event designators may be used:

`!` Refers to the previous event.

Example: The history reference `!!` repeats the previous command.

n Refers to event number *n*.

–*n* Refers to the current event minus *n*.

str Refers to the most recent event starting with *str*.

`?`*str*[`?`]

Refers to the most recent event containing *str*.

The second `?` delimiter may be omitted if a newline character comes immediately after *str*.

Without the event designator, the history reference applies

- to the most recent command line entered, or
- to the command line referenced by an immediately preceding history reference (if any).

The event designator may be legally omitted only if the ! which begins the history reference is not followed by a space, a tab, a newline, an equal sign or a left parenthesis.

Example:

`!:1 !$ !:3:s/f/file/` are legal history references.

`!` is not a legal history reference; to repeat the execution of the last command line entered, you must use `!!`.

Word designators

An event designator may optionally be followed by a word designator, separated by a colon. The colon can be omitted if the word designator begins with ^, $, *, – or %.

A word designator is used to select one or more words from the command line. The following word designators are permitted:

`#` The entire command line typed so far.

`0` The first input word (i.e. the command name).

n The *n*th argument.

`^` The first argument (equivalent to 1).

`$` The last argument.

`%` The word matched by the most recent `?`*str* search.

n–*m* Inclusive range from *n*th to *m*th argument.

–*m* Abbreviation for 0–*m*.

n`*` Abbreviation for *n*–`$`.

n– Like *n*`*`, but without the last argument.

`*` All the arguments, or a null value if there is just one word in the event.

Modifiers

After the event designator (or optional word designator), you can add one or more modifiers. Each of these modifiers must be preceded by a colon (except for the `g` modifier).

Caution:

It is not possible to use modifiers on the output of command substitutions.

There are two restrictions on modifier usage with variable substitutions: not all of the modifiers are available, and only one modifier is allowed per substitution.

The following modifiers are available:

`h` (head) Removes the last path name component, leaving the head.

`t` (tail) Removes all leading path name components, leaving the tail.

`r` (root) Removes a trailing suffix of the form `.`*xxx* leaving the basename.

`e` Removes the entire path name, except for the suffix.

`s/`*l*`/`*r*`[/]`

Substitutes the string *r* for *l*. Unless preceded by the `g` modifier, the modification is applied only to the first string that matches *l*; an error is reported if no string matches.

Any character can be used as the delimiter in place of /; the delimiter character is automatically defined by the fact that it immediately follows `s`. If you wish to use the delimiter character in string *l* or *r*, you will need to escape it with a backslash.

The ampersand character & in the replacement string *r* is a metacharacter; it is replaced by the text from string *l*. The & can be escaped with a backslash.

If *l* is a null string, a previous string is used; this string is either the *l* string of the previous `s` command or the string *str* from the previous `!?`*str* command.

The rightmost delimiter may be omitted if a newline immediately follows *r*.

`&` Repeats the previous substitution.

`g` (global) This modifier may be prefixed to one of the modifiers above, i.e. `h`, `t`, `r`, `e`, `s`, and `&` (e.g. `gr` or `gs` or `g&`). The change is then applied to the first occurrence of a match in *each* word. (Caution: the change is *not* applied to all matching strings!)

`p` (print) The new command created by the history substitution is placed on the history list and printed on the screen, but not executed.

`q` Quotes the substituted words, escaping further substitutions.

`x` Like `q`, but splitting into words at each space character, tab or newline.

Braces {...}

Braces can be used to insulate a history reference from subsequent characters that are not part of the command:

`!{...}`*string*

See *Example 6* at the end of the C shell description.

Quick substitution

`^`*l*`^`*r*`[^]`

This is the short form of the history reference `!:s^`*l*`^`*r*`[^]` (substitute string *r* for *l* in the previous command line or in the command line designated by the last history reference). See *Example 7* at the end of the C shell description.

The `histchars` variable can be used to change the ^ (caret) for quick substitutions to some other character.

Command aliasing

Frequently used command sequences can be abbreviated by defining aliases for them with the C shell built-in

`alias`.

`alias` *alias-name alias-definition*

The C shell enters the alias and the associated alias definition in a list of aliases, which can be displayed with the `alias` command (without arguments). Entries can be deleted from the list with `unalias`. (See *Example 8* at the end of the C shell description.)

Arguments can be specified in the alias definition as follows:

- Arguments which are constants are specified directly.

  Example:

  ```
  alias cpr cc –c prog.c
  ```

  Invoking `cpr` executes `cc –c` *prog.c*.

- For arguments which are to be supplied on the command line, you must specify placeholders in the alias definition in the form of history references. Each history reference must be escaped with a backslash to ensure that the shell expands it when expanding the alias name, not when executing the `alias` command.

  Example:

  ```
  alias gpw grep \!^ /etc/passwd
  ```

  Invoking `gpw anne` executes `grep anne /etc/passwd`.

  No history reference is required if the variable argument does not need to appear within the alias definition but can simply appear after it.

  Example:

  ```
  alias lld ls –ld
  ```

  Invoking `lld` *dir* executes `ls –ld` *dir*.

The shell resolves aliases in command lines as follows: after parsing the command line into individual commands, the shell checks the first word in each command. If this word is a valid alias, the shell reads the corresponding alias definition and expands any history references in it. The original command line with the alias is interpreted as though it were the previously entered command. If there are no history references in the alias definition, the shell appends the argument list of the original command line to the alias definition (see *Examples 8* and *9* at the end of the C shell description).

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

```
alias lm 'ls –l \!* | more'
```

(See *Example 10* at the end of the C shell description.)

If the alias definition for an alias *name1* contains some other alias *name2*, then the definition of *name2* must not include the alias *name1*, as this would result in a loop during alias substitution. Such loops are detected by the shell, and cause an error message.

An alias definition may also contain the name of the alias itself, but only as the first word. This does not result in a loop, as the shell stops after the first alias substitution in this case and does not search for further aliases (see *Example 11* at the end of the C shell description).

I/O redirection

The expression forms below can be used to redirect a command's standard input, standard output, and standard error. File names following the metacharacters are subjected to variable, command, and file name substitutions separately from the rest of the command. These substitutions are not carried out on the string *word* (see `<<`*word*).

If you wish to redirect the standard output and standard error separately, you must invoke a subshell as follows:

(*command* `>` *outputfile*) `>&` *errorfile*

`<` Redirects the standard input.

`<<`*word*

Reads the standard input, up to a line that is identical with *word*, and places the resulting lines in a temporary file. Unless *word* is escaped or quoted, variable and command substitutions are performed on these lines. This means that variables are replaced by their values, and commands enclosed in backquotes are replaced by their output. The command for which the standard input was redirected is then invoked with the temporary file as its standard input.

`> >! >& >&!`

Redirects the standard output to a file. If the file does not exist, it is created. If it does exist, it is overwritten, and its previous contents are lost.

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

`>> >>! >>& >>&!`

These forms are like the> forms above, but append the standard output to the end of an existing file rather than overwriting it.

If the `noclobber` variable is set and the file does not exist, the shell will report an error, unless one of the ! forms is used. The & forms append both standard output and standard error output to the file.

Variable substitution

C shell variables are composed of a name and a value.

A variable name is a string of up to 20 letters, digits, and underscores (_), starting with a letter or an underscore. A variable's value is a space or tab-separated list of zero or more words. If you enclose the word list in parentheses when defining the variable, you can access individual elements of the list with $*var*[*index*] or ${*var*[*index*]} (see *Shell parameters* and *Example 12* at the end of the C shell description).

The C shell makes a distinction between environment variables and shell variables: environment variables are exported automatically, i.e. made known to invoked processes; shell variables are not (see **Environment variables and predefined shell variables**). Unless explicitly stated otherwise, the information contained in the present section on **Variable substitution** is applicable to both types of variable.

Creating, displaying, and deleting variables

The C shell built-in command `set` can be used to display existing variables and to define new ones; `unset` is used to delete variables.

Variables used by the C shell

Some variables are predefined and are maintained or used by the C shell. For instance, the `argv` variable contains an image of the shell's argument list.

Of the variables used by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not. A typical example of such a variable is `verbose`; when this variable is set to any value (including an empty value), each command you enter is echoed on the screen after history substitution.

Predefined variables are described in detail in the section on **Environment variables and predefined shell variables**.

Referencing the value of a variable

To refer to a variable's value you precede its name with $.

$name

The expression *$name* represents a shell parameter. The shell replaces parameters in the command line with the current value of the corresponding variable. You can also select specific words from the variable value or display other information on the variable (see *Shell parameters*).

If the name of a shell parameter is followed by a letter, a digit, or an underscore _, the shell will interpret this character as a part of the name. Braces can be used to insulate the shell parameter from other characters in an input-line word.

Example:

The command

`echo $file`

can be used to display the value of the variable `file` (e.g. `/usr/tables/list`). The command below appends the ending 00 to the file name:

`mv $file ${file}00`

If the braces were omitted, the shell would search for a variable named `file00`, which may not have been defined.

The shell will normally report an error if you refer to an undefined variable.

Numerical variable values can be operated on as numbers. With numeric operations, an empty variable value is interpreted as zero; the second and subsequent words of multiword values are ignored.

Escape characters and variable substitution

Variable substitution can be suppressed by preceding the dollar sign with a backslash, except within double quotes ("..."), where variables are always substituted. Variable substitution is suppressed within single quotes ('...'). The dollar sign is also escaped if followed by a space character, a tab or a newline.

Shell parameters in double quotes are replaced by their values; metacharacters within the value are treated as if enclosed in single quotes. Multiword values are expanded to a single string of words with embedded space characters. For example, if the value of the variable `name` is `Anne Miller`, the argument "`$name`" would be equivalent to "`Anne Miller`".

Note the difference when the `q` modifier is specified with the history reference: if the variable has a multiword value, the shell parameter is not expanded to a single string but to a list of space-separated words, each of which is quoted independently to prevent subsequent command and file name substitution.

Timing of variable substitution

Variable substitution normally takes place after the input line has been analyzed, aliases resolved, and I/O redirections applied. Exceptions to this are:

- variables appearing in I/O redirections and in input lines in `<<`*word* redirections (substituted at the time the redirection is made),
- variables appearing in backquoted commands (see **Command substitution**).

Command and file name substitution is not performed until values have been substituted for variables. These substitutions are suppressed when:

- the command or file name pattern is enclosed within double quotes,
- the `noglob` variable is set (suppresses file name substitution only), or
- the history reference is quoted with the history modifier `q`.

Shell parameters

Shell parameters are introduced by the $ character. The shell replaces any such parameter in the command line by the current value of the corresponding variable.

The shell parameter forms recognized by the C shell are:

$*var*
${*var*}

These shell parameters are replaced by the value of the *var* variable. With multiword values, the result is a space-separated list of words.

If *var* is an environment variable (see **Environment variables and predefined shell variables**), its value is returned (but no history modifiers can be used).

$*var*[*index*]
${*var*[*index*]}

These shell parameters select only the indicated words from the value of *var*. The value of the variable must be enclosed in parentheses when the variable is defined (see the C shell built-in `set`).

index may consist of or evaluate to a single number, a range defined by two numbers separated by "–" (e.g. *num1–num2*), or an asterisk (*). Words in the value of the variable are indexed starting from 1; an asterisk selects all words. If *index* specifies a number that exceeds the number of words in the value of

the variable, the shell will report an error. A range may be specified in short form as follows:

–num2 short form for 1*–num2*;

num1– is the short form for *num1–total*, where *total* is the total number of words in the value of the variable.

It is not an error for a range to be empty if the second argument is omitted or lies within the permitted range 1 to `$#`*var*.

`$#`*name*
`${#`*name*`}`
These shell parameters return the number of words in the value of the variable.

`$0` The shell parameter `$0` evaluates to the name of the file from which command input is being read. An error occurs if this file name is not known.

`$`*n*
`${`*n*`}`
Equivalent to `$argv[`*n*`]` (see the predefined shell variable `argv`); *n* is a number greater than or equal to 1.

`$*` Equivalent to `$argv[*]` (see the predefined shell variable `argv`).

`$?`*var*
`${?`*var*`}`
Returns 1 if *var* is set, 0 if it is not.

`$?0` Returns 1 if the current input file name is known, 0 if it is not.

`$$` Returns the process ID of the (parent) shell.

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

The modifiers `e`, `h`, `q`, `r` and `t` plus `gh`, `gt` and `gr` (see **History substitution**) can be applied to the shell parameters listed below. If {...} (braces) are used, the modifiers must appear within the braces.

`$`*var* or `${`*var*`}`□
`$`*var*`[`*index*`]` or `${`*var*`[`*index*`]}`□
`$#`*name* or `${#`*name*`}`□
`$0`□
`$`*n* or `${`*n*`}`□
`$*`

Command substitution

A command that is enclosed in backquotes (`...`) is executed by a subshell. Its standard output is split into separate words at each space character, tab and newline; null words are discarded. Within double quotes, only the newline character acts as a delimiter; space and tab characters are preserved.

The resulting text replaces the backquoted string on the current command line.

A final newline character is ignored. It is therefore possible for a command substitution to yield a partial word, even if the standard output of the command has returned a complete line.

Command and file name substitutions (i.e. the substitution of command output for backquoted commands and matching file names for patterns) are applied selectively to the arguments of built-in commands. These substitutions are only performed for evaluated expressions; portions of expressions that are not evaluated are not expanded.

For external commands, file name expansion of the command name is done separately from that of the argument list; expansion is done in a subshell, after I/O redirection has taken place.

File name substitution

Unquoted words which

- contain any of the characters *, ?, [, or {

- or begin with ~ (tilde)

are treated as patterns. The shell expands these patterns as follows (see also *Example 13* at the end of the C shell description):

- Patterns featuring *, ?, or [are replaced by an alphabetically sorted list of file names which match the pattern and identify existing files. The shell reports an error if there is no matching file name for such a pattern.
- The pattern {...} is a short form for a list of strings. These strings need *not* be names of existing files. The list is not sorted.
- The patterns ~ and *~login_name* represent short forms for a home directory.

The metacharacters have the following functions:

`*` matches any (zero or more) characters.

`?` matches any single character.

`[`*s*`]` matches any single character contained in the enclosed string *s*.

`[`*c1*–*c2*`]`
matches any single character in the range from *c1* to *c2* inclusive in the ASCII collating sequence.

{*s1*,*s2*,...}
expands to any of the strings (or file name matching patterns) in the comma-separated list *s1*, *s2*,... etc. Unlike the *, ?, [*s*] and [*c1*–*c2*] expressions above, the expansion of {*s1*,*s2*,...} is not sorted.

~ ashort form for your home directory, as defined by the value of the `home` variable.

~login_name
ashort form for the home directory of the specified user, as defined by the entry for that user in the password file `/etc/passwd`.

If the above metacharacters are enclosed within single or double quotes or preceded by a backslash, they are interpreted as normal characters, without any special meaning.

To match a path name with a component that begins with a dot (e.g. `/usr/anne/.cshrc`), the pattern must include the dot explicitly. A slash must also be matched explicitly.

The characters { and } (left and right braces) and the string {} (an empty pair of braces) are not expanded but are passed undisturbed.

You will find more information on file name substitution in the section on **Command substitution**.

Expressions and operators

A number of C shell built-in commands accept expressions, in which the operators are similar to those of the C programming language and follow the same rules of precedence. These expressions typically appear in the `@`, `exit`, `if`, `set` and `while` commands, and are often used to regulate the flow of control for executing commands (see **Control flow**).

The individual components of an expression are separated by space or tab characters. Null or missing values are treated as 0. The result of an expression is always a string, which may be a decimal number.

The following C shell operators are grouped in order of precedence; operators in the same line have the same precedence.

`(...)` Grouping.

`!` Logical negation.

`* / %` Multiplication, division, remainder (of integer division).

These operators are right associative, which may lead to unexpected results. When in doubt, you should group combinations explicitly with parentheses.

`+ –` Addition, subtraction.

These operators are also right associative.

`<< >>` Bitwise shift left, bitwise shift right.

`< > <= >=`
: Less than, greater than, less than or equal to, greater than or equal to.

`== != =~ !~`
: Equal to (`==`), not equal to (`!=`), file name substitution pattern match (`=~`), file name substitution pattern mismatch (`!~`). These operators compare their arguments as strings.

  The operators `=~` and `!~` each check whether or not a string on the left matches a file name substitution pattern on the right. This reduces the need for `switch` statements in shell scripts when pattern matching between strings is all that is required.

`&`
: Bitwise AND.

`^`
: Bitwise exclusive OR (XOR)

`|`
: Bitwise inclusive OR.

`&&`
: Logical AND.

`||`
: Logical OR.

The operators: `==`, `!=`, `=~`, and `!~` compare their arguments as strings; other operators use numbers.

The following file inquiries are also available:

`-r` *file*
: Returns "true" (i.e. 1) if you have read permission for *file*; otherwise returns "false" (i.e. 0).

`-w` *file*
: True if you have write permission for *file*.

`-x` *file*
: True if you have execute permission for *file* (or search permission for a directory).

`-e` *file*
: True if *file* exists.

`-o` *file*
: True if you own the file.

`-z` *file*
: True if *file* is of zero length (i.e. empty).

`-f` *file*
: True if *file* is a plain file.

`-d` *file*
: True if *file* is a directory.

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

There is also an inquiry to determine whether a command has completed successfully:

{ *command* }
: If *command* runs successfully, the expression evaluates to "true" (i.e. 1). Otherwise, it evaluates to "false" (i.e. 0).

  Note that commands typically return 0 when successful and some other value if they encounter problems. If you want to inspect the command exit status directly, you should use the value of the `status` variable rather than this expression.

Control flow

The shell contains a number of built-in commands to regulate the flow of control in scripts, and within limits, from the terminal. These commands operate by forcing the shell either to execute a loop (e.g. with `foreach`, `while`) or to skip a part of the input under certain conditions (in a `switch` or `if` branch, etc.). See the section on **C shell built-in commands**, *Control-flow statements*.

If the shell's input is not seekable and a loop is being read, the input is buffered. The shell can go to certain positions within the buffer to do the rereading implied by the loop. To the extent that this allows, backward `goto` commands will succeed on nonseekable inputs.

Command execution

If the command is a C shell built-in, the shell executes it directly. Otherwise, the shell searches for a file by that name with execute permission. If the command name contains a forward slash, the shell treats it as a path name, and searches for it at the appropriate location. If the command name does not contain a slash, the shell normally searches through its hash table to find the file. The hash table is an internal table maintained by the shell. It contains the path names of all the executable files in the directories listed in the `path` variable. The current directory constitutes an exception: the files contained in it do not appear in the hash table, but are checked separately. If the file is neither in the hash table nor in the current directory, the shell issues a message indicating that the file was not found.

An interactive C shell will normally recreate the hash table in the following cases:

- after it has read the `.cshrc` file,
- after the `path` variable has been reset,
- when you invoke the C shell built-in `rehash`.

The `rehash` command should be used to recompute the internal hash table whenever the contents of directories listed in the `path` variable have changed (e.g. to account for new commands added).

The hash table is used to speed up the search for the file to be executed. It can be disabled with the `csh` options `-c` or `-t`, or with the C shell built-in `unhash`. If this is done, the shell will check the `path` variable and search for the required executable file in all the directories listed in this variable.

Special case:

If the command to be executed is a shell script that has no / (slash) in its name, and there is an alias for the word `shell`, the expansion of the shell alias is prepended (without modification) to the command line. The system attempts to execute the first word of this special (late-occurring) alias, which should be a full path name. Any remaining words of the alias's definition, together with the text of the input line, are treated as arguments.

If the file that is found has proper execute permissions, the shell uses `fork(2)` to spawn a new process and passes it to the kernel together with its arguments (using the `execve(2)` system call). The kernel then attempts to overlay the new process with the chosen program.

If the file is an executable binary [in `a.out` format, see `a.out(4)`], the overlay succeeds, and the kernel begins executing the new process.

If the file is a text file (shell script) and its first line begins with `#!`, the next word is treated as the path name of a shell (or some other command) to be used to execute the script, and subsequent words on the first line are interpreted as options for the named shell. The kernel invokes (overlays) the indicated shell, using the name of the script as an argument.

If the file is a shell script and its first line does not begin with `#!`, the kernel cannot overlay the file (the `execve(2)` call fails). The C shell then spawns a new shell, as follows:

- it invokes a C shell (`csh`) if the first line of the file begins with a # character,
- otherwise, it invokes a standard Bourne shell (`sh`).

The file is then executed in the newly spawned shell.

Signal handling

The C shell normally ignores `QUIT` signals. Background processes are immune to signals generated from the keyboard, including hangups (`HUP`). Other signals have the values that the C shell inherited from its environment.

The shell's handling of interrupt and terminate signals within scripts can be controlled by the C shell built-in `onintr`.

Login shells catch `TERM` signals. If the shell is not a login shell, `TERM` signals are passed on to child processes. In no case are interrupts allowed when a login shell is reading the `.logout` file.

Job control

The C shell assigns a numbered job to each command sequence or pipeline, to keep track of those commands that are running in the background or have been stopped with `TSTP` (terminal stop) signals (typically <CTRL-Z>). When a command sequence or pipeline is started in the background using the & metacharacter, the shell displays a line with the job number in square brackets, and a list of associated process IDs. For example:

```
[1] 1234
```

In this example, the job has been assigned job number 1 and is associated with the top level process with process ID 1234.

You can use the `jobs` built-in to view the current list of jobs. The job most recently stopped (or moved into the background if none are stopped) is referred to as the *current* job, and is marked "+" in the list. The previous job is marked "–"; when the current job is terminated moved to the foreground, this job takes its place (becomes the new current job).

To stop a running job, you must send it a stop signal (typically <CTRL-Z>). Once the shell has reported that the job has stopped, and has displayed the prompt, you can change the status of the job. The built-in commands `bg`, `fg`, `kill`, `stop` and `%` are provided for this purpose. For example, you can move the job to the background with `bg`, then execute some other commands, and then return it to the foreground with `fg`.

Caution:

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

Shell built-in commands cannot be stopped and restarted.

Command sequences in the form

command1; *command2*; *command3*

are problematic. If you suspend *command2*, the shell never executes *command3*. This is especially noticeable if the expansion results from an alias. You can avoid this problem by placing the sequence in parentheses (...) to have it executed in a subshell.

Jobs can be referenced as shown below when calling the `bg`, `fg`, `kill`, `stop` and `%` commands:

`% %+ %%`
The current job.

`%–` The job previous to the current job.

`%`*jobnumber*
The job with the specified jobnumber.

Example:
The command `kill –9 %3` terminates job number 3.

`%`*string*
The job for which the command line begins with the specified string. *string* must identify the command line uniquely.

Example:
The command `fg %vi` brings a stopped `vi` job to the foreground.

`%?`*string*
The job for which the command line includes *string*. *string* must identify the command line uniquely.

A job running in the background stops when it attempts to read from the terminal. Background jobs can normally produce output, but this can be suppressed with the `stty tostop` command. A background job will then be stopped as soon as it attempts to produce output.

Job status change reporting

While running interactively, the C shell tracks the status of each job and reports whenever a job finishes or becomes blocked. It normally displays a message to this effect immediately before issuing a prompt, so as to avoid disturbing the appearance of your input. If the `notify` variable is set, the shell reports status changes

immediately, instead of waiting until a prompt is about to be issued. The shell built-in `notify` can also be used for this purpose: `notify` instructs the shell to report status changes in certain jobs immediately (see **C shell built-in commands**, `notify`).

C shell built-in commands

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

The following built-in commands are available in the C shell:

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

`alias` [*name* [*def*]]
Assigns definition *def* to alias *name*. *def* is a list of words which may contain history substitution metasyntax. These metacharacters must be escaped with a backslash. *name* is not allowed to be `alias` or `unalias`.

def not specified: The alias *name* is displayed together with its current definition.

name[*def*] not specified: All aliases are displayed.

See **Command aliasing**.

`bg` [*job* ...]
(background) Runs the specified jobs in the background. Jobs can be referenced as described in the section on **Job control**.

job not specified: The current job is run in the background (see **Job control**).

`cd` [*dir*]
`chdir` [*dir*]
Changes the shell's working directory to *dir*.

If *dir* is a relative path name not found in the current directory, `csh` will search for it in the directories listed in the `cdpath` variable. If *dir* is the name of a shell variable whose *value* starts with a slash, the current directory is changed to the directory named by that *value*.

dir not specified: Changes to the user's home directory.

`dirs` [`–l`]
Prints the directory stack. The list is arranged from left to right, with the topmost directory on the stack (the current directory) shown on the left.

`–l` Produces an unabbreviated listing of all directories. Even if the `~` notation is used to specify a directory (see **File name substitution**), that directory will be listed in full.

`echo` [`–n`] *list*
The words in *list* are written to the C shell's standard output, separated by space characters. If the `–n` option is not used, the output is terminated with a newline.

`–n` No newline character is output at the end of the list.

`eval` *argument* ...
The C shell executes the commands which appear in the argument list. The arguments are thus evaluated twice by the shell:

- once when the shell processes the `eval` command line.
- again when the shell executes the processed command-line arguments as commands.

See the Bourne shell built-in `eval`.

`exec` *command*
Executes the specified command in place of the current shell, which terminates.

`exit` [`(`*expression*`)`]
The shell exits with the value of *expression* (see **Expressions and operators**).

`(`*expression*`)` not specified: The shell exits with the value of the `status` variable.

`fg` [*job* ...]
(foreground) Brings the specified jobs into the foreground. Jobs can be referenced as described in the section on **Job control**.

job not specified: The current job is moved to the foreground (see **Job control**).

`glob` *wordlist*
Substitutes matching file names for the patterns specified in *wordlist* and then prints the word list. `glob` operates similarly to `echo`, but does not recognize backslashes as escapes for metacharacters. Words are delimited by NULL characters in the output.

`hashstat`
Prints a statistics line indicating how effective the internal hash table has been at locating commands in the file tree (and at avoiding `exec` calls in the process). In searching for commands, an `exec` is attempted for each component of the `path` variable

- for which the hash function indicates a possible hit, and
- which does not begin with a slash.

See **Command execution** for more details.

`history` [–hr] [*n*]
Displays the history list; if a number *n* is given, only the *n* most recent events are displayed.

`–h` Displays the history list without leading event numbers. This is used to produce files suitable for `source –h` command (see the C shell built-in `source`).

`–r` Reverses the order in which the history list is printed out so that the most recent event is shown first rather than the oldest event.

`jobs` [–l]
Lists the active jobs under job control.

`–l` Lists process IDs, in addition to the normal information.

`kill` [–*signal*] [*pid* ...] [*job* ...]
`kill` [–l]
Send the specified signal, or by default the `TERM` (terminate) signal, to the specified processes or jobs. If the signal being sent is `TERM` (terminate) or `HUP` (hangup), then the job or process is sent a `CONT` (continue) signal as well. You must specify at least one process or job; both may not be omitted.

signal
Signal number or signal name.

pid Process ID.

job Jobs can be specified as described in the section on **Job control**.

`–l` Lists the signal names that can be sent with `kill`.

`limit` [–h] [*resource* [*max-use*]]
Limits consumption of the specified resource by the current process or any process it spawns to a maximum of *max-use*.

If *max-use* is omitted, the current limit for *resource* is printed; if *resource* is omitted, all limits are displayed.

resource
The following values can be specified for *resource*:

`cputime`
Maximum amount of CPU time allowed per process.

`filesize`
Largest single file allowed.

`datasize`
Maximum size of data area (including stack) allowed for the process.

`stacksize`
Maximum stack size allowed for the process.

`coredumpsize`
Maximum size of a core dump file.

max-use
max-use can be given as a number, optionally followed by a letter to indicate the scaling factor, or as a value in minutes and seconds (for `cputime`):

resource	*max-use*	Meaning of *max-use*
`cputime`	*n*	*n* seconds
	n`m`	*n* minutes
	mm`:`*ss*	*mm* minutes and *ss* seconds
	n`h`	*n* hours
`filesize`, `datasize`,	*n*	*n* kilobytes
`stacksize` and	*n*`k`	*n* kilobytes
`coredumpsize`	*n*`m`	*n* megabytes

`-h` Causes hard limits to be used instead of the current limits. Hard limits impose a ceiling on values for the current limits. Only the system administrator may raise the hard limits.

`login` [*login_name*]
`login` [`-p`]
Terminates a login shell and invokes the `login` command. The `.logout` file is not processed. If *login_name* is omitted, `login` prompts you for a login name.

`-p` Preserves the current environment (variables).

`logout`
Terminates a login shell.

`nice` [*n*] [*command*]
Increments or decrements the process priority value for the shell or for the specified command by *n*.

The higher the priority value, the lower the priority of a process, and the slower it runs. The range of process priority values is from -20 to 20. Values of *n* outside this range set the value to the lower, or to the higher boundary, respectively.

`+`*n* Increments the process priority value by *n*.

`-`*n* Decrements the process priority value by *n*. This argument can be used only by the system administrator.

n not specified: The process priority value is set to 4.

command
Command for which the process priority value is to be set. *command* is run in a subshell with the set value, and with the same restrictions as apply to a command in the control-flow statement `if (`*expr*`)` *command*.

command not specified: `nice` sets the process priority value for the current shell.

`nohup` [*command*]
Runs the specified command, ignoring `HUP` signals. *command* is always run in a subshell, and with the same restrictions as apply to a command in the control-flow statement `if (`*expr*`)` *command*. If no command is specified, `HUP` signals are ignored throughout the remainder of a script.

Processes that are started in the background with & are always run with `nohup`.

`notify` [*job* ...]

Instructs the C shell to report changes in the status of the specified jobs immediately instead of waiting until a prompt is about to be issued. Jobs can be referenced as described in the section on **Job control**.

job not specified:

Changes in the status of the *current job* are reported immediately. The job most recently stopped or moved into the background (if none are stopped) is referred to as the current job.

`onintr` [–]
`onintr` [*label*]

Controls the actions the shell takes in response to interrupts.

No argument specified:

`onintr` restores the default action of the shell on interrupts, i.e. the shell terminates shell scripts and waits for command input from the terminal.

– The shell ignores all interrupts.

label The shell executes a `goto` *label* when an interrupt is received or a child process terminates because it was interrupted.

`popd` [+*n*]

Pops the directory stack (i.e. discards its top element), and changes to the new top directory with `cd`.

The elements of the directory stack are numbered from 0, starting at the top.

+*n* The *n*th entry in the stack is discarded.

`pushd` [*dir*]
`pushd` [+*n*]

Pushes a directory onto the directory stack (i.e. adds it to the top of the stack).

No argument specified:

Exchanges the top two stack elements and changes to the new top directory with `cd`.

dir Pushes the current working directory onto the stack and changes to *dir* with `cd`.

+*n* Rotates the *n*th entry to the top of the stack and changes to it with `cd`.

`rehash`

Recomputes the internal hash table of the contents of directories listed in the `path` variable. See also **Command execution**.

`repeat` *count command*

Repeats the specified command *count* times.

command is subject to the same restrictions as apply to the control-flow statement `if (`*expr*`)` *command*.

`set` [*var*[=*value*]]
`set` *var*[*n*]=*word*

No argument specified:

`set` displays the values of all shell variables. Multiword values are displayed as a parenthesized list.

var `set` *var* assigns an empty (null) value to the variable *var*.

var=*value*

`set` *var*=*value* assigns *value* to the shell variable *var*. The equal sign may be surrounded by white space (blanks or tabs).

value can be any of the following:

word A single word (or a string enclosed in single or double quotes).

`(`*wordlist*`)`

A space-separated list of words enclosed in parentheses. This assigns a multiword value to *var*.

If *value* includes file name substitution metacharacters or backquoted commands, it is subjected

to file name and command substitution before being assigned to the variable.

var[*n*]=*word*

`set` *var*[*n*]=*word* replaces the *n*th word in the multiword value of a shell variable (*var*) with *word*. The variable *var* must be an existing variable that already has an *n*th word in its value.

The square brackets are mandatory; they do not represent optional syntax elements in this case. The equal sign may be surrounded by white space (blanks and tabs).

As with `set` *var*=*value*, command and file name substitution is performed on the value before the assignment is made.

`setenv` [*VAR* [*word*]]

No argument specified:

`setenv` displays the values of all environment variables.

VAR `setenv` *VAR* assigns an empty (null) value to the environment variable *VAR*.

By convention, environment variables are normally given uppercase names.

VAR word

`setenv` *VAR word* assigns the value *word*, which must be either a single word or a quoted string, to the environment variable *VAR*.

The most commonly used environment variables, `USER`, `TERM`, and `PATH`, are automatically imported to and exported from the `csh` variables `user`, `term`, and `path`; there is no need to use `setenv` for these. In addition, the shell sets the `PWD` environment variable from the `csh` variable `cwd` whenever the latter changes.

`shift` [*var*]

The components of *argv*, or of *var* if supplied, are shifted to the left, discarding the first component.

It is an error for *argv* or *var* not to be set, or to have a null value.

`source` [`–h`] *file*

The `source` command executes shell scripts in the current shell, i.e. without creating a subshell. `source` reads commands from the named *file* and executes them. `source` commands may be nested; however, if they are nested too deeply, the shell may run out of file descriptors. An error in any of the shell scripts terminates all nested `source` commands.

`–h` Places commands from *file* on the history list without executing them.

`stop` *job* ...

Stops the specified background jobs. Jobs can be referenced as described in the section on **Job control**.

`suspend`

Suspends the current shell as if it had been sent a stop signal with <CTRL-Z>. This is most often used to stop shells started by `su`.

`time` [*command*]

`time` executes the specified command and prints a summary of the time used by that command.

command not specified:

`time` prints a summary of time used by the current C shell and its child processes.

`umask` [*value*]

`umask` sets or displays the current file-creation mode mask. This mask defines the permissions to be assigned to all the files and directories that you subsequently create in the current shell or in any of its subshells (see the Bourne shell built-in `umask`).

value A three-digit octal number specifying the file-creation mode mask. *value* is XORed with the permissions of 666 for files and 777 for directories to arrive at the permissions for new files.

Common values for the file-creation mode mask include 002 or 022, which set the following permissions:

002 sets	for files:	`rw-rw-r--`
	for directories:	`rwxrwxr-x`
022 sets	for files:	`rw-r--r--`
	for directories:	`rwxr-xr-x`

`unalias` *pattern*
: Discards all aliases that match *pattern* from the alias list. *pattern* can be defined as any word and may include file name substitution metacharacters (see **File name substitution**).

`unhash`
: Disables the internal hash table (see **Command execution**).

`unlimit` [`–h`] [*resource*]
: Removes the limits from the specified resource.

resource
: See `limit`.

resource not specified: All resource limits are removed.

`–h` Removes the corresponding hard limits. This option may only be used by the system administrator.

`unset` *pattern*
: Removes all shell variables whose names match *pattern*. *pattern* can be defined as any word and may include file name substitution metacharacters (see **File name substitution**).

The command `unset *` removes all shell variables, but may have unwanted side-effects.

`unsetenv` *VAR*
: Removes environment variable *VAR* from the environment. You cannot specify a pattern for *VAR* as you can with `unset`.

`wait` Causes the shell to wait for all background jobs to finish (or for an interrupt) before prompting.

`%`[*str*][`&`]
: `%`*str* moves the job identified by `%`*str* to the foreground.

`%`*str*`&` leaves the job identified by `%`*str* to continue running in the background.

str can be any of the following: +, %, –, *jobnumber*, *string*, or `?`*string* (see **Job control**).

str not specified:

The current job is moved to the foreground or left running in the background.

The command `%`[*str*] is equivalent to `fg %`[*str*].

The command `%`[*str*]`&` is equivalent to `bg %`[*str*].

`@` [*var*=*expr*]
`@` [*var*[*n*]=*expr*]
: *No argument specified*:

`@` displays the values of all shell variables.

var=*expr*
var[*n*]=*expr*

`@` *var*=*expr* assigns the shell variable *var* the value that *expr* evaluates to (see **Expressions and operators**).

`@` *var*[*n*]=*expr* replaces the *n*th word in the multiword value of a shell variable (*var*) with the value that *expr* evaluates to. *var* and its *n*th component must already exist.

The square brackets are mandatory; they do not represent optional syntax elements in this case.

If *expr* contains the characters >, <, & or |, then these characters or the entire expression must be placed within parentheses (...).

The operators *=, +=, etc., are available as in C. The space separating the variable name from the assignment operator is optional.

In addition, you can use the special postfix operators ++ and —— to increment or decrement *var* by 1.

Control-flow statements

Control-flow statements are built-in commands of the C shell. The C shell provides the following control-flow statements:

`break` Exit a `foreach` or `while` loop.

`breaksw`
Exit from a `switch` statement.

`continue`
Continue a `foreach` or `while` loop at the next iteration.

`foreach ... end`
Process a list in a loop.

`goto` Jump to a labeled position.

`if` Test for a condition and execute a command.

`if ... then ... else if ... then ... else ... endif`
Test for conditions and execute associated commands.

`switch ... case ... breaksw ... default ... breaksw ... endsw`
Conditional branches.

`while ... end`
Loop with exit condition.

`break` A `break` statement is used within a `foreach` or `while` loop in order to terminate the loop. `csh` resumes execution after the loop's `end` statement. Any remaining commands on the current line are executed. This allows multilevel breaks to be written as a list of `break` commands, all on one line, enabling you to break out of nested loops.

`breaksw`
`breaksw` can be used within a `switch` statement to exit from the switch. Execution is resumed after the associated `endsw` statement.

`continue`
`continue` is used within a `foreach` or `while` loop; `csh` terminates the current iteration of the loop and resumes execution at the next iteration.

`foreach` *var* `(`*wordlist*`)`
...
`end`
Each word in *wordlist* is sequentially assigned as a value to the variable *var*, and the commands between `foreach` and `end` are executed after each assignment.

The statements `foreach` *var* `(`*wordlist*`)` and `end` must appear alone on separate lines.

There may be `break` and `continue` statements (see above) in lines between the `foreach` and `end` statements.

If the shell reads the `foreach` statement from a terminal (as opposed to a shell script), it will prompt for commands with a query (?) before executing the commands in the `foreach` loop. You may then specify additional commands in response to the query. It is only when you enter `end` that the shell will execute the other commands in the `foreach` loop. The commands that you enter in response to the ? prompt are not placed on the history list.

`goto` *label*
If the *label* contains file name substitution metacharacters or backquoted commands, it is first subjected to file name and command substitution to yield a label (see **File name substitution** and **Command substitution**). The shell then looks back through its input as far as it can, searching for a line of the

form

label : □

(possibly preceded by space or tab characters). Execution continues after the indicated line. It is an error to jump to a label appearing in the body of a `while` or `foreach` loop.

`if (`*expr*`)` *command*

If *expr* evaluates to true, *command* is executed. *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Variable substitution on *command* occurs at the same time as it does for *expr*.

If *command* contains I/O redirection metacharacters, the redirection will occur even if *expr* is false.

`if (`*expr1*`) then`
...
`else if (`*expr2*`) then`
...
`else`
...
`endif`

If *expr1* evaluates to true, commands up to the first `else` are executed, and control is passed to the statement following `endif`.

If *expr1* is false, *expr2* is evaluated.

If *expr2* is true, the commands between `else if` and the second `else` are executed, and control is passed to the statement following `endif`.

If *expr2* is false, commands between the second `else` and `endif` are executed.

Any number of `else if` pairs are allowed, but only one `else` statement. The `endif` statement is only needed once, but it is mandatory.

The statement `if (`*exprX*`) then` must appear alone on a separate line, or after an `else`. The words `else` and `endif` must be the first non-whitespace characters on a line (i.e. can only be preceded by blanks or tabs).

`switch (`*string*`)`
`case` *label*`:`
...
`breaksw`
...
`default:`
...
`breaksw`
`endsw`

If *string* contains file name substitution metacharacters or backquoted commands, it is first subjected to file name and command substitution (see **File name substitution** and **Command substitution**). The file name substitution metacharacters *, ? and [...] may be used in the `case` labels; variables in these labels are also substituted.

Each *label* is then successively matched against the specified *string*. If a match is found, the commands following that label are executed. If none of the `case` labels match before a `default` label is found, execution begins with the commands after the `default` label. If no label matches and there is no `default` label, execution continues after the `endsw`.

The `breaksw` statement terminates the `switch` statement and continues execution after the `endsw`.

If *string* matches a label and there is no `breaksw` between that label and the next, commands of subsequent labels are also executed (as in fall-through in the C programming language).

The keywords `switch`, `case`, `default`, and `endsw` must appear at the beginning of separate lines; the `default` label should be specified after the `case` labels.

`while (`*expr*`)`

...
end

Commands between the while statement and the matching end statement are executed repeatedly as long as *expr* continues to evaluate to true.

The while (*expr*) and end statements must appear alone on separate lines.

There may be break and continue statements (see above) in lines between the while and end statements.

If the shell reads the while statement from a terminal (as opposed to a shell script), it will prompt for commands with a query (?) before executing the commands in the while loop. You may then specify additional commands in response to the query. It is only when you enter end that the shell will execute the other commands in the while loop. The commands that you enter in response to the ? prompt are not placed on the history list.

Environment variables and predefined shell variables

Unlike the standard shell, the C shell maintains a distinction between environment variables, which are automatically exported to processes it invokes, and shell variables, which are not. Both types of variables are treated similarly in respect of variable substitution.

The C shell sets the variables argv, cwd, home, path, prompt, shell, and status upon initialization. In addition, it copies the environment variable USER into the shell variable user, TERM into term, and HOME into home, and copies each back into the respective environment variable whenever the shell variables are reset. PATH and path are handled similarly; you only need to set path once in the .cshrc or .login file. The environment variable PWD takes its value from the shell variable cwd, changing whenever the latter changes.

The set path command can be used to remove duplicate path names from the value of path. These may occur because a shell script or a .cshrc file includes a command like set path=(/usr/local/usr/hosts $path) to ensure that the named directories are added to the path name list.

The following shell variables have predefined meanings:

argv Argument list containing the list of command-line arguments supplied to the current invocation of the shell. This variable governs the values of the positional parameters $1, $2, etc.: $argv corresponds to $1 and so on.

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

cwd Contains the full path name of the current directory.

echo Setting this variable causes command input to be echoed on standard error after all substitutions and just before command execution. See the csh options –x and –X.

fignore
List of file name suffixes to be ignored when attempting file name completion. This list typically consists of the single suffix .o (see **File name/login name completion**).

filec Setting the filec variable enables an interactive C shell to complete partially entered file names and login names. In this case the ESC and EOF (<CTRL-D>) characters have special significance when typed in at the end of a terminal input line:

ESC If the string preceding ESC begins with a tilde (~), the shell attempts to extend the string to *~login*, where *login* is an existing login name. If more than one name matches, the terminal bell is rung.

If the string preceding the ESC character does not begin with a tilde, the shell attempts to extend the string to match a file name from the current directory. If more than one name matches, the terminal bell is rung.

EOF If a string is followed by the EOF character, the shell lists all matching login names or file names in the current directory that begin with the specified string. It then prompts once again, supplying

the incomplete command line typed in so far. You can then enter one of the listed login names or file names.

(See **File name/login name completion** for more details.)

`hardpaths`
If this variable is set, path names in the directory stack are resolved so as to contain no symbolic-link components. This variable is extremely useful, since symbolic links can confuse the C shell.

`histchars`
A two-character string in which the first character defines the initial character for history substitutions (usually the exclamation point !), and the second defines the metacharacter to be used for quick substitutions (usually the caret ^ as in ^*l*^*r*[^]). See *Quick substitution* in the section on **History substitution** for more details.

`history`
The number of lines saved in the history list. Any number may be specified; however, a very large number may use up all the memory that can be allocated by the C shell.

If this variable is not set, only the most recent command is saved.

`home` Contains the user's home directory.

When a home directory is referenced by means of a tilde in a line of command input, the shell refers to the value of this variable to determine the appropriate directory (see **File name substitution**).

`ignoreeof`
If this variable is set, the shell ignores EOF characters coming from terminals. This prevents the shell from being accidentally terminated by someone typing <CTRL-D>.

`mail` A list of files which the C shell checks for incoming mail. If the first word of the value of `mail` is a number *n*, it specifies a mail checking interval of *n* seconds. If no number is specified, `mail` files are checked every 5 minutes.

`nobeep`
Setting this variable stops the terminal bell ringing when multiple matches of file names or login names are available as possible extensions for a partially entered name (see **File name/login name completion**).

`noclobber`
Setting this variable restricts output redirection as follows: > redirections can only be made to new files; >> redirections can only be made to existing files (see **I/O redirection**).

`noglob`
Inhibits file name substitution. This is useful in shell scripts once the required file names have been obtained and no further expansion is desired.

`nonomatch`
Returns the file name substitution pattern, rather than an error, if the pattern cannot be matched. Patterns with an incorrect syntax still result in errors.

`notify`
If this variable is set, the shell reports changes in the status of jobs immediately, rather than waiting until just before issuing a prompt (see **Job control**).

`path` List of directories to be searched for the appropriate file to execute a command (see **Command execution**). If the value of `path` is the null string, only the current directory is searched. The default for `path` is typically:

(. `/usr/ucb /usr/bin`).

If the `path` variable is not set, the shell will only execute commands named by their absolute path names.

`path` is initialized from the environment variable `PATH`. If the value of `path` changes, the shell updates the environment variable `PATH` by copying the new value of `path` back to it.

An interactive C shell will normally hash the contents of the directories listed in the `path` variable (i.e.

add them to the hash table) in the following cases:

- after it has read the `.cshrc` file,
- after the `path` variable has been reset,
- when you invoke the C shell built-in `rehash`.

The `rehash` command should be used to recompute the internal hash table whenever the contents of directories listed in the `path` variable have changed (e.g. to account for new commands added). See **Command execution**.

`prompt`

The string an interactive C shell prompts with.

An exclamation point (!) in the `prompt` string is replaced by the current event number when the prompt is displayed (see **History substitution**). Note that in the `set` call defining the prompt you must escape the exclamation point with a backslash to prevent the shell from starting a history substitution (see **EXAMPLES**, **History substitution**).

The default prompt of an interactive shell is *hostname*`%` for ordinary users, or *hostname*# for the system administrator.

Non-interactive C shells leave the `prompt` variable unset. To reduce startup time for non-interactive shells, aliases and other commands that are only useful interactively can be placed in the `.cshrc` file conditionally on the basis of the following test:

```
if ($?prompt == 0) exit
```

(If the `prompt` variable is not set, i.e. if the shell is non-interactive, `exit` is executed.)

`savehist`

The number of lines from the history list that are saved in `~/.history` when the user logs out.

Large values for `savehist` slow down the C shell during startup.

`shell` The file in which the C shell resides.

This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. See **Command execution**.

`status`

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

`time` Setting this variable causes the shell to display a summary of the resources used by each command after the command terminates.

`time` can be given a null value or a value comprising one or two components:

- If no value is supplied (null), the summary takes the form:

  `%U %S %E %P %X %D %I %O %F %W` (see below).
- The first component of the value can be given as a number *n*, where *n* is the reporting threshold in CPU seconds. This means that the summary includes only those commands that have used at least *n* CPU seconds.
- The second component of the value is a string that defines the output format for the summary [see the format string for `printf(3S)`]. The string may contain normal characters as well as tags. Normal characters are printed unchanged. A tag is a percent sign (%) followed by a single letter; it is replaced in the output by the value it represents. The following tags may be specified:

  `%D` Average amount of unshared data space used (in kilobytes).

  `%E` Elapsed time for the command (real time).

  `%F` Page faults.

  `%I` Number of disk block input operations.

  `%K` Average amount of unshared stack space used (in kilobytes).

`%M` Maximum real memory used during execution of the process.

`%O` Number of disk block output operations.

`%P` Total CPU time, i.e. `%U` plus `%S`, as a percentage of `%E`.

`%S` Number of seconds of CPU time consumed by the kernel on behalf of the user's process (i.e. time spent in system mode).

`%U` Number of seconds of CPU time devoted to the user's process (i.e.time spent in user mode).

`%W` Number of swaps.

`%X` Average amount of shared memory used (in kilobytes).

`verbose`
If this variable is set, the shell displays each command after history substitution. See the `csh` options `–v` and `–V`.

C shell scripts

Like the standard Bourne shell (`sh`), the C shell (`csh`) also provides the essential elements of a programming language. This allows you to write your own procedures known as C shell scripts.

A C shell script consists of one or more command lines. There are, in particular, a number of built-in commands to regulate the flow of control in scripts. These commands operate by forcing the shell either to execute a loop or to skip a part of the input under certain conditions (see the section on **C shell built-in commands**, *Control-flow statements*).

Shell scripts are usually held in files called script files, which must have both read and execute permissions. Script files do not need to be compiled. They are normally started with the command `csh` *filename*. When you invoke a shell script, the shell spawns a subshell, which reads, interprets, and executes each line in the *file*. Once all the commands in this file have been executed, the subshell exits and the parent shell returns.

C shell scripts can also be started with the shell built-in `source`. This command executes shell scripts in the current shell, i.e. without spawning a subshell.

The hash character **#** introduces a comment that continues to the end of the input line. All characters that make up a comment are ignored by the shell when it processes the script.

If the first line of a shell script begins with `#!`, the next word is treated as the path name of a shell (or some other command) to be used to interpret the script. Subsequent words on the first line are treated as options for the named shell. The kernel invokes (overlays) the indicated shell, using the name of the script as an argument.

If the first line of the script does not begin with `#!`, `csh` then spawns a new shell, as follows:

- it invokes a C shell (`csh`) if the first line of the file begins with a # character,
- otherwise, it invokes a standard Bourne shell (`sh`).

The script is then executed in the newly spawned shell.

CONSTRAINTS

The C shell is subject to the following constraints:

- Words can be no longer than 1024 characters.
- The system limits argument lists to 1.048.576 characters.
- The maximum number of arguments to a command for which file name expansion applies is 1706.
- Command substitutions may expand to an argument list containing no more than 1.048.576 characters.
- To detect looping, the shell restricts the number of alias substitutions on a single line to 20.

ERROR MESSAGES

`You have stopped jobs.`

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

FILES

`~/.cshrc`
: File executed by the C shell at startup.

`~/.login`
: File executed by login C shells after `.cshrc` at login.

`~/.logout`
: File executed by login C shells at logout.

`~/.history`
: File in which the history list is saved at logout for use at the next login.

`/usr/bin/sh`
: Standard shell. `sh` is a Bourne shell. Shell scripts not starting with a # are executed by `sh`.

`/tmp/sh*`
: Temporary file for << redirections (see **I/O redirection**).

`/etc/passwd`
: Password file containing, among other things, the home directories needed by the C shell for evaluation of patterns in the form *~login_name* (see **File name substitution**).

EXAMPLES

Unless indicated otherwise, the default prompt *hostname*`%` is abbreviated to `%` in the following examples.

File name/login name completion

Examples 1 and *2* are based on the assumption that the `filec` variable is set.

Example 1: *File names*

The current directory contains the files `chaosnet`, `class`, `cmd`, `list.old` and `list.new`.

If you enter:

`% vi ch` <ESCAPE>

the shell will complete your input line with:

```
vi chaosnet
```

However, if you enter:

`% vi l` <ESCAPE>

the shell will ring the terminal bell, since more than one file name matches (`list.old` and `list.new`).

If you enter the EOF character instead of <ESCAPE>, all matching file names will be listed:

```
% vi l <CTRL-D> □
list.old list.new □
% vi l
```

Example 2: *Login names*

The input line:

`% cd ~ro` <ESCAPE>

will be completed as:

```
cd ~root
```

if `ro` represents a unique abbreviation. This takes you `root`'s home directory.

History substitution

The examples in this section are based on the assumption that the C shell prompt has been defined in the following way:

```
set prompt="\!% "
```

The prompt the C shell displays will then be the current event number, followed by a percent sign and a space

(see the shell variable `prompt`).

The `history` variable should also be set to a sufficiently high value (such as 10).

Example 3

Event designators

In the command sequence that follows, you first use the `vi` editor to create a C source file called `prog.c` and then compile it using the C compiler `cc`. As you have made a typing error, `cc` reports a syntax error. You therefore edit the file again, eliminate the error, and recompile your source with `cc`:

```
1% vi prog.c□
  ...□
2% cc –O prog.c –o form□
  ...□
3% vi prog.c□
  ...□
4% cc –O prog.c –o form□
5%
```

History substitution allows you to avoid entering the same commands twice:

```
1% vi prog.c□
  ...□
2% cc –O prog.c –o form□
  ...□
3% !v   or   3% !vi□
vi prog.c□
  ...□
4% !c   or   4% !cc   or   4% !?form□
cc –O prog.c –o form□
5%
```

As you can see, the history reference `!v` or `!vi` was used in event 3. In this case, `v` or `vi` is the event designator; it references the most recent command entered starting with `v` (or `vi`). That command was the command `vi prog.c`. The C shell prints this command line on the screen and then executes it.

The history references `!c` and `!cc` (in event 4) are based on the same principle and execute the command `cc –O prog.c –o form` in the same way.

The history reference `!?form`, by contrast, achieves the same result by designating the most recently entered command containing the string `form` (at any position).

Besides using strings, you may also reference commands by their event numbers. If you do not know which number is assigned to the command you want in the history list, you can use the C shell built-in `history` to print the list. Event numbers can be specified in absolute terms or relative to the current event:

```
1% vi prog.c□
  ...□
2% cc –O prog.c –o form□
  ...□
3% history□
   1 vi prog.c□
   2 cc –O prog.c –o form□
   3 history□
4% !1       or   4% !–3□
vi prog.c□
  ...□
5% !2   or   5% !–3□
cc –O prog.c –o form□
6%
```

Example 4

Word designators

Most recently entered command	Input line with history references	Input line after history substitution
`grep ^anne /etc/passwd`	`grep !*`	`grep ^anne /etc/passwd`
	`grep !:1 file`	`grep ^anne file`
	`grep !^ file`	
	`grep ^cathy !:2`	`grep ^cathy /etc/passwd`
	`grep ^cathy !$`	

Example 5

Modifiers

Most recently entered command	Input line with history references	Input line after history substitution
`ls –l /home/anne/plan`	`ls –ld !$:h`	`ls –ld /home/anne`
	`ls –ld !$:t`	`ls –ld plan`
`vi prog.c`	`cc !$ –o !$:r`	`cc prog.c –o prog`
	`cc !$ –o !$:s/.c/`	
`cc prog.c –o prog`	`!:gs/prog/list`	`cc list.c –o list`
	`!:s/prog/list`	`cc list.c –o prog`
`cat f1 f2 f3`	`cat !:1:s/d/file/ !:3:&`	`cat file1 file3`

If you are using complex history references, you should first use the `p` parameter to check whether history substitution produces the desired result before executing the command. In the case of the above example, you could do this as follows:

```
1% cat f1 f2 f3□
  ...□
2% cat !:1:s/f/file/ !:3:&:p□
cat file1 file3□
3% !!□
cat file1 file3□
  ...□
4%
```

Example 6

Braces

Most recently entered command	Input line with history references	Input line after history substitution
`ls –ld ~paul`	`!ls`	`The last command beginning with ls`
	`!{l}a`	`ls –ld ~paula`
	`!!a`	
	`!{!}a`	

Example 7

Quick substitution

Most recently entered command	Input line with history references	Input line after history substitution
`grep ^cathy /etc/passwd`	`^ht^th` `!:s^ht^th` `!:s/ht/th`	`grep ^cathy /etc/passwd`

Aliases

Example 8

Simple example

```
% alias lld ls –ld
% alias
...
lld (ls –ld)
...
```

If you enter:

```
% lld /usr
```

the shell will read the alias definition `ls –ld` for the alias `lld` and append the argument `/usr` to it. This will result in the following command:

```
ls –ld /usr
```

The command:

```
% unalias lld
```

removes the alias entry `lld ls –ld` from the alias list. The entire alias list can be deleted with:

```
% unalias *
```

Example 9

History references in alias definitions

```
% alias gpw grep \!^ /etc/passwd
```

Due to the backslash, the shell will not expand the history reference when executing the `alias` command:

```
% alias
gpw (grep !^ /etc/passwd)
```

If you now enter:

```
% gpw anne
```

the shell will read the alias definition `grep !^ /etc/passwd` for the alias `gpw` and expand the history reference `!^`. For this history substitution, the shell will interpret the command `gpw anne` as the most recent event, i.e. with the same effect as if you had entered the commands:

```
gpw anne
grep !^ /etc/passwd
```

in that order. This means that the history reference `!^` designates the argument `anne`, so the resulting command would be:

```
grep anne /etc/passwd
```

Here are two more complex examples:

```
% alias vcc 'vi \!^; cc \!^ –o \!^:r'
% alias
vcc vi !^; cc !^ –o !^:r
```

The command:

```
% vcc prog.c
```

would then produce the command sequence:

```
vi prog.c; cc prog.c –o prog
```

When you call the `alias` command, you must enclose the alias definition in single quotes, as the shell would otherwise interpret the semicolon as the end of the `alias` command instead of seeing it as a part of the alias definition:

```
% alias lm 'ls –l \!* | more'
```

The command:

```
% lm dir1 dir2
```

would then return the pipeline:

```
ls –l dir1 dir2 | more
```

Example 10

Nested alias definitions

```
% alias clm 'cd \!^; lm'
%
%
% alias lm 'ls –l | more'
```

The command:

```
% clm dir
```

would then produce the command sequence:

```
cd dir; ls –l | more
```

Example 11

Alias definition containing the alias itself

```
% alias ls ls –l
```

The command:

```
% ls dir
```

would then expand to:

```
ls –l dir
```

Variable substitution

Example 12

The following example shows how C shell variables can be treated like arrays in C. You can initialize the variables with a word list, address the individual elements using subscripts, and assign values to them.

```
% set color=(red pink green)
% echo $color
red pink green
% echo $#color
3
% echo $color[3] $color[1] $color[2-3]
green red pink green
% set color[2]=blue
% echo $color
red blue green
```

To begin with, the shell variable `color` is assigned the word list `red pink green` as its value. The second command prints the value of `color`. The command `echo $#color` returns the number of words contained in the value of `color`. The last three commands show how to address and modify individual value components.

File name substitution

Example 13

The current directory contains the files `list`, `plan`, `plan1`, `plan2`, `plan3`, and `planning`.

The subdirectory `prog` contains the files `ls.c`, `newls.c`, `oldls.c` and `pgm.c`.

Pattern	Expanded to
`plan*`	`plan plan1 plan2 plan3 planning`
`plan?`	`plan1 plan2 plan3`
`plan[12]`	`plan1 plan2`
`plan[1-3]`	`plan1 plan2 plan3`
`prog/*ls*`	`prog/ls.c prog/newls.c prog/oldls.c`
`prog/{oldls,ls,main}.c`	`prog/oldls.c prog/ls.c prog/main.c`
`./{plan?,list}`	`./plan1 ./plan2 ./plan3 ./list`

Note the order in the above table: when the braces are expanded, the strings are not sorted, but when the other metacharacters are expanded, the matching file names are sorted in alphabetical order.

Furthermore, as you can see from the second-last line in the table, the shell expands the braces directly, i.e. without comparing the resulting strings with the names of existing files. Consequently, the list obtained by expanding the pattern includes a path name `prog/main.c`, even though no such file exists.

If your home directory is `/home/anne`, and the home directory of a user named john is `/home/john`, then the patterns `~` and `~john` would be expanded as follows:

Pattern	Expanded to
`~/version5/prog`	`/home/anne/version5/prog`
`~john/prog`	`/home/john/prog`
`~john/prog/*.c`	`/home/john/prog/cls.c /home/john/prog/ls.c`

SEE ALSO

`login`(1), `sh`(1), `access`(2), `exec`(2), `fork`(2), `pipe`(2), `a.out`(4), `terminfo`(4), `ascii`(5), `environ`(5).

User's Reference Manual

csplit(1)

NAME

`csplit -` context split

SYNOPSIS

`csplit` [*option ...*] *file argument ...*

DESCRIPTION

`csplit` splits the contents of a file or the text it reads from standard input into smaller sections and writes all or some of these sections to separate output files. The original file is left unaltered.

The way in which `csplit` divides a file and the sections for which output files are created are specified in the command line arguments.

If `–n` is not specified, `csplit` creates a maximum of 100 output files per call.

OPTIONS

No option specified:

The output files are named `xx00`, `xx01`, and so on.

For each output file that it creates, `csplit` writes a character count on standard output.

Any files that have already been created are removed if an error occurs.

`–f` *name*

The output files are called `name00`, `name01`, etc.

`–f` not specified:

The output files are named `xx00`, `xx01`, and so on.

`–k` Files that have already been created are retained if an error occurs.

`–n` *number*

The current number of the output files comprises *number* digits, whereby 1 £ *number* £ 9.

Example: For `–n 4`, the output files are called `xx0000`, `xx0001` etc.

`–n` not specified:

The current number consists of 2 digits.

`–s` The output of a character count is suppressed.

`––` If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file Name of the input file.

If you use a dash (–) as the name for *file*, `csplit` reads from standard input.

argument

You can specify several arguments, each of which references a particular line in the input file. These lines represent the points at which `csplit` is to split the file into sections. Each dividing line becomes the first line of a new section. If you specify *n* arguments, `csplit` divides the file into *n*+1 sections. These sections contain the following lines:

Section	Contents
0	All lines from the start of the input file up to but not including the line referenced by the first argument.
1	All lines from the first dividing line up to but not including the line referenced by the second argument.
...	...

n	All lines from the line referenced by the *n*th argument to the end of the input file.

`csplit` usually writes each section to a separate output file.

This does not apply when the argument `%`*regular_expression*`%`[+*number*][–*number*] is used (see below). The last section (section *n*) is always written to an output file.

The arguments you specify are processed by `csplit` in the order in which you list them. To begin with, the first line of the input file is the current line. After an argument has been processed, the line referenced by this argument becomes the current line. The line referenced by the next argument must lie in the range between but not including the current line and the end of the input file. Thus the line referenced by the second argument must come after the line referenced by the first argument.

argument can be specified as follows:

`/`*regexp*`/`[+*number*][–*number*]

An argument in the form `/`*regexp*`/` references the next line after the current line that matches the specified regular expression. The section from the current line up to but not including the line that matches the regular expression is written to an output file. The line matching the regular expression now becomes the current line.

The +*number* or –*number* offset shifts the dividing line *number* lines after (+) or before (–) the line that matches the regular expression. The line that is *number* lines after (+) or before (–) the line matching the regular expression thus becomes the current line.

Simple regular expressions [see `expressions(5)`] are recognized. If the argument contains blanks or shell metacharacters [see `specialchar(5)`], you must either escape every such character with a backslash \ or enclose the whole argument in single quotes '...'. The regular expression must not contain any newline characters.

`%`*regexp*`%`[+*number*][–*number*]

An argument in the form `%`*regexp*`%` references the next line after the current line that matches the specified regular expression. The line that matches the regular expression becomes the current line. `csplit` in this case *does not* create an output file for the relevant section.

If the +*number* or –*number* offset is also specified, the current line will be the line that is *number* lines after (+) or before (–) the line containing the regular expression.

Simple regular expressions [see `expressions(5)`] are recognized. If the argument contains blanks or shell metacharacters [see `specialchar(5)`], you must either escape every such character with a backslash \ or enclose the whole argument in single quotes '...'. The regular expression must not contain any newline characters.

num This argument references the line with line number *num*. `csplit` writes the section from the current line up to but not including the *num*th line to an output file. The *num*th line then becomes the current line.

{*n*} This argument is an abbreviation for *n* arguments of the previous type (see above) and means: "repeat the preceding argument *n* times", where *n* is an integer greater than 1.

The {*n*} argument can be entered after any of the above-mentioned arguments, with a blank to separate them.

Thus if it follows an argument in the form `/`*regexp*`/`+*number*][–*number*] or `%`*regexp*`%`[+*number*] [–*number*], this argument will be repeated *n* times.

Example:

```
'/regexp/'{2}
```

is an abbreviation for

```
'/regexp/''/regexp/''/regexp/'
```

If {*n*} follows an argument of the *num* type, the file will be split *n* times, from the *num*th line onward, into sections of *num* lines each.

Example:

```
100 {2}
```

is an abbreviation for

```
100 200 300
```

ERROR MESSAGES

argument – out of range

The line referenced by the specified *argument* lies outside the permissible range. The legal range is from, but not including, the current line to the end of the file.

'*xx*'file limit reached ...

You have specified so many arguments that `csplit` had to create more output files than the value specified for the `–n` option allows.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_CTYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The file `book` contains a text that is subdivided into three chapters. The first chapter is preceded by a preface; an appendix follows the last chapter. Each chapter begins with the title "CHAPTER ..."; the title of the appendix is "APPENDIX".

You now wish to put the preface, the individual chapters, and the appendix into separate files. The output files are to be named `chap00`, `chap01`, etc.

```
$ csplit –f chap book '/CHAPTER/' '/CHAPTER/' '/CHAPTER/' '/APPENDIX/'
1636
15124
32743
20344
2576
$ ls
book
chap00
chap01
chap02
chap03
chap04
```

The file `chap00` contains the preface and consists of 1636 characters. The appendix is located in the file `chap04`.

The same results could also have been obtained by abbreviating the `csplit` call as follows:

```
$ csplit –f chap book '/CHAPTER/' {2} '/APPENDIX/'
 .
 .
```

You can now edit the sections separately, and later you can join them again using `cat`:

```
$ cat chap0[0–4] > book
```

Example 2

The input file `file` is to be split into sections every hundred lines. To do this, you enter:

```
$ csplit file 100 {98}
```

The argument `{98}` stands for 98 arguments: 200 300 ... 9900.

If `file` contains 9900 or more lines, `csplit` creates 100 output files. The first output file `xx00` includes line 1 to 99 (inclusive); the last output file, `xx99`, contains the rest of `file` from line 9900 onward.

If `file` contains fewer than 9900 lines, `csplit` issues the error message `{98} – out of range` and terminates. If you include option `–k` in the call, the files already created are retained.

```
$ csplit –k file 100 {98}
```

If `file` contains only 9830 lines, for example, then `xx98` is the last output file created and includes lines 9800 to 9830.

Example 3

The file `prog.c` contains a C source program. The program includes a `main` function and a maximum of 20 further functions. In accordance with C conventions, each function ends with a right brace at the beginning of a line (in column 1). Right braces within a function are not located in the first column of a line.

Each function is now to be written to a separate file. To do this, you enter:

```
$ csplit –k prog.c '%main(%''/^}/+1'{19}
```

If the program contains exactly 20 functions in addition to the `main` function, `csplit` splits the file into 22 sections.

Section 0 contains all lines from the beginning of the file up to but not including the start of the `main` function. This section will *not* be written to an output file (argument `%main(%`).

Section 1 contains the `main` function and is written to the output file `xx00` (argument `/^}/+1`).

Functions 1 to 19 are similarly written to separate output files in succession (argument `{19}`). The final section, i.e. section 22, contains the rest of the input file (which in this case is function 20) and is written to the output file `xx20`.

If the program contains fewer than 20 functions, `csplit` will terminate at the last function and issue the error message `{19} – out of range`. Since the `–k` option has been set, the created files will, however, be retained.

SEE ALSO

`ed(1)`, `sed(1)`, `sh(1)`, `split(1)`.

User's Reference Manual

cut(1)

NAME

`cut -` cut out selected fields of each line of a file

SYNOPSIS

Format 1

`cut –b` *list* [`–n`] [`––`] [*file ...*]

Format 2

`cut –c` *list* [`––`] [*file ...*]

Format 3

`cut –f` *list* [`–d` *char*] [`–s`] [`––`] [*file ...*]

DESCRIPTION

`cut` reads an input text from files or from standard input one line at a time and cuts out specific columns (**Format 1**) or fields (**Format 2**) from these lines. The selected columns or fields are written to standard output.

OPTIONS

Format 1: Cutting out columns

`cut –b` *list* [`–n`] [`––`] [*file ...*]

`cut –c` *list* [`––`] [*file ...*]

`–b` *list*

(byte) `cut` cuts out the columns specified in *list* from each input line and writes them to standard output. A column is exactly one byte wide.

list is a list of numbers or a range. The individual elements in the list must be separated by commas and arranged in ascending order. A range of columns given as *n1–n2* includes all numbers from *n1* up to, and including, *n2*.

The following abbreviations are permissible in range definitions:

–n for 1 – *n*

n– for *n* – "last column"

Example:

`1,3,5`
: `cut` cuts out columns 1, 3, and 5.

`1–3,5`
: `cut` cuts out columns 1, 2, 3, and 5.

`–3,5` is the abbreviation for 1 – 3, and 5.

`3–` is the abbreviation for 3 – "last column".

`–n` Characters comprising a number of bytes are not split. The upper and lower range definition limits are reduced internally until the range includes only complete characters.

`–c` *list*

(column) The columns specified in *list* are cut out from each input line and written to standard output. A column is exactly one character wide.

list is list of numbers or number ranges, as described for the `–b` option.

`––` If *file* begins with a dash (–), the end of the command line options must be marked with `––`.

file Name of the input file.

You may name more than one file.

file not specified: `cut` reads from standard input.

Format 2: Cutting out fields

`cut –f` *list* [`–d` *char*] [`–s`] [`––`] [*file* ...]

`–f` *list*

(field) `cut` cuts out the fields specified in *list* from each input line and writes them on standard output.

A field comprises all characters that are located between two field delimiters. Two successive field delimiters mark an empty field. The field delimiter defaults to the tab character, but you can redefine it in the `–d` option.

The output fields are separated from one another by one field delimiter each. Input lines with no field delimiters are output in full (unless the `–s` option is set). This is generally useful for table subheadings.

list is specified as described under **Format 1**.

`–d` *char*

The character given as *char* serves as the field delimiter.

If *char* is a blank or a shell metacharacter [see `specialchar(5)`] it must be enclosed in single quotes: `–d` '*char*'.

`–d` not specified:

The field delimiter defaults to the tab character.

`–s` Lines with no field delimiters are suppressed.

`–s` not specified:

Lines with no field delimiters are output in full.

`––` If *file* begins with a dash (–), the end of the command line options must be marked with `––`.

file Name of the input file.

You may name more than one file.

file not specified: `cut` reads from standard input.

ERROR MESSAGES

`ERROR: line too long`

A line can have no more than 1023 characters or fields. Alternatively, the newline character may be missing.

`ERROR: bad list for c/f option`

Missing `–c` or `–f` option or incorrectly specified *list*. No error is reported if a line has fewer fields than required for the *list*.

`ERROR: no fields`

The *list* is empty.

`ERROR: no delimiter`

The delimiter character missing from the `–d` option.

`ERROR: cannot handle multiple adjacent backspaces`

Adjacent backspaces cannot be processed correctly.

`WARNING: cannot open` *file*

The named *file* cannot be read or does not exist. If multiple files have been specified, processing will continue on the other files.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise

undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Print the first 72 characters of each input line:

```
$ cut –c 1–72 file
```

Example 2

Output the first 72 bytes of each input line under the condition that multi-byte characters are not split:

```
cut –b 1–71 –n file
```

Example 3

The first and third fields of the /etc/passwd file (i.e. the login name and user ID) are cut out and output. The individual fields in this file are separated by means of a colon.

```
$ cut –f 1,3 –d: /etc/passwd
```

Example 4

The names of customers whose bills are due on the first day of any month are to be filtered out of the invoice file of a mail order house along with the amount due from each of them.

The file is structured as follows:

```
Homewood        Milwaukee         10,000        06.01.91
Mackenzie       Detroit            7,000        07.06.91
Macnamara       Boston             8,000        05.01.91
Tinniswood      Atlanta              450        06.20.91
```

The fields in the table are separated by exactly one tab and padded with blanks.

```
$ grep '\.01\.91' invoice | cut –f 1,3 > names
$ cat names
Homewood        10,000
Macnamara        8,000
```

Explanation: grep finds all lines containing the string .01.91 and writes them to standard output. cut receives these lines as input, selects the first and third fields, and writes its output to the file names.

If you would then like each customer name in the names file to be preceded by the date and then sorted accordingly, you could enter:

```
$ grep '\.01\.91' invoice | cut –f 4 > date
$ paste date names | sort
05.01.91        Macnamara         8,000
06.01.91        Homewood         10,000
```

Explanation: The first command line writes the date associated with the selected customer names to the file named date. The paste command pastes the lines in the date and names files horizontally, separating them with the default tab character. The sort command then sorts the output lines in ascending order of date.

SEE ALSO

awk(1), grep(1), paste(1), sh(1).

date(1) - dumpmsg(1)

NAME

`date` - print date and time or set system clock

SYNOPSIS

Format 1

`date` [–`u`] [+*format*]

Format 2

`date` [–`u`] [*new_date*]

Format 3

`date` [–`a` [–]*ssss*.*fff*]

DESCRIPTION

`date` writes the current date and time on standard output (**Format 1**). The form in which `date` outputs dates and times depends on the value of the environment variable `LC_TIME` or, if that is undefined or null, on the value of `LANG`. If both are undefined or null, or if the required database is not available, or if one of the NLS variables has an invalid value, `date` acts as if in a non-internationalized system, printing dates and times in American format.

If you are the system administrator, you can also use `date` to set (**Format 2**) and to adjust (**Format 3**) the system clock; but you should not do this unless you are in single-user mode and all file systems have been unmounted. Changing the system clock may otherwise produce inconsistencies in the file systems.

Mode of operation

The system operates in coordinated universal time UTC (Universal Time Coordinated; synonymous with Greenwich Mean Time/GMT). `date` takes care of conversions to and from UTC to local time. If the environment variable `TZ` is defined, it is used to determine the timezone and to convert UTC to local time [see `sh`(1) , "Standard shell variables"].

OPTIONS

Format 1: Print date and time

`date` [–`u`] [+*format*]

No argument specified:

`date` prints the current date and time in a format governed by the current locale.

–`u` Displays the current date and time in Greenwich Mean Time (GMT).

If the –`u` option is set, the +*format* specification is ignored.

+*format*

The *format* argument defines the output format for `date`. If *format* contains blanks or tabs or other shell metacharacters which you do not want the shell to interpret, you should enclose *format* in single quotes: +'*format*'.

format is essentially similar in format to the first argument of the `printf` function in C or `awk` [see `awk`(1) , `printf`(1) , and the C function `printf`(3S)].

The output of `date` can be formatted by means of field descriptors in the *format* argument. Field descriptors take the form of a percentage sign followed by a letter and are replaced in the output by their values. All characters that are not part of a field descriptor are copied to the output unmodified. The output is always terminated with a newline character.

The following overview lists all legal field descriptors. Although the field descriptors `%h` and `%b` are essentially identical, both descriptors have been retained for compatibility reasons.

`%n`	newline character
`%t`	tab character

%c	date and time in the default format of the current locale
%f	identical to %c
%C	century (first two digits of the year, 00-99)
%D	date in the format %m/%d/%y
%x	date in the format of the current locale
%y	year (last two digits of the year, 00-99)
%Y	year (all four digits of the year)
%m	month (01 to 12)
%h	month (in letters, abbreviated) in the format of the current locale
%b	identical to %h
%B	month (in letters, in full) in the format of of the current locale
%W	week number of the year (00 to 53, with Monday as the first day of the week). All days before the first Monday in January belong to week 0.
%V	week number of the year (01 to 53, with Monday as the first day of the week). The first week in January is counted as week 1 if it contains at least 4 days. Otherwise this week is counted as week 53 of the previous year (in accordance with ISO 8601).
%U	week number of the year (00 to 53, with Sunday as the first day of the week). All days before the first Sunday in January belong to week 0.
%j	day of year (001 to 366)
%d	day of month (01 to 31)
%e	day of month (1 to 31, single-digit numbers preceded by a blank)
%a	abbreviated weekday name in the format of the current locale
%A	full weekday name in the format of the current locale
%w	day of week (0 to 6, Sunday = 0)
%u	day of week (1 to 7, Monday = 1)
%R	time in the format %H:%M
%T	time in the format %H:%M:%S
%X	time in the format of the current locale
%r	time in a.m./p.m. notation, as %I:%M:%S %p
%H	hour (00 to 23)
%I	hour (01 to 12)

%p	string containing ante-meridian or post-meridian indicator (a.m./p.m. affix) in the format of the current locale.
%M	minute (00 to 59)
%S	second (00 to 61)
%Z	timezone name, or no output if no timezone exists (governed by the value of the environment variable TZ, see sh(1) , "Standard shell variables").

If an alternative representation is defined in your local environment (e.g. before and after Christ) you

can call it using modified field descriptors. Modified field descriptors are in the form `%E`*letter* or `%O`*letter*. The modified field descriptors that can be used are listed in the overview below:

If no alternative representation has been defined, all modified field descriptors output the value of the current unmodified field descriptors.

`%Ec`	date and time in the alternative format
`%EC`	name of the time period (e.g. "year") in the alternative representation
`%Ex`	date in the alternative format
`%EX`	time in the alternative format
`%Ey`	year in which the time period in the alternative representation begins
`%EY`	year in the alternative representation
`%Od`	day of month in alternative numeric representation
`%Oe`	day of month in alternative numeric representation
`%OH`	hour (24-hour clock) in alternative numeric representation
`%OI`	hour (12-hour clock) in alternative numeric representation
`%Om`	month in alternative numeric representation
`%OM`	minutes in alternative numeric representation
`%OS`	seconds in alternative numeric representation
`%Ou`	weekday in the alternative format (Monday = 1)
`%OU`	week of the year in alternative numeric representation (same as for `%U`)
`%OV`	week of the year in alternative numeric representation (same as for `%V`)
`%Ow`	weekday in the alternative format (Sunday = 0)
`%OW`	week of the year in alternative numeric representation (same as for `%W`)
`%Oy`	year in the alternative format

Format 2: Reset system clock

`date` [`–u`] [*new_date*]

This format is for the system administrator only.

This format can be used to reset the date.

`–u` Sets the date and time in Greenwich Mean Time (see option `–u` under **Format 1**).

new_date

`date` sets the date and time of the system clock to the specified value. The following can be entered for [*mmdd*]*HHMM*[[*cc*]*yy*]:

mm	*dd*	*HH*	*MM*	
month (01 – 12)	day (01 – 31)	hour (00 – 23)	minute (00 – 59)	century minus 1

mmdd not specified:

`date` assumes that the specified time *HHMM* refers to the current date.

cc not specified:

If the two-digit year notation *yy* is in the range `70` to `99`, the 20th century is assumed; if it is in the range `00` to `37`, the 21st century is assumed. All other entries are rejected.

For *yy* only the digits >69 and <38 may be entered. If the year number specified is <38, this is beyond the year 2000.

yy not specified:

The current year is assumed.

Format 3: Adjust system clock

`date` [–a [–]*ssss*.*fff*]

This format is for the system administrator only.

This format can be used to adjust the system clock.

–a [–]*ssss*.*fff*

Adjusts the system clock time by *ssss*.*fff* seconds, where *fff* represents fractions of a second. This adjustment can be positive or negative (–). The system's clock will be sped up or slowed down until it has changed by the specified amount (*ssss* may not exceed 2146).

ERROR MESSAGES

Format 2

`date: no permission.`

You have tried to set the system clock as a normal user. This privilege is reserved for the system administrator.

`date: bad conversion.`

You have attempted to set the system clock but have called `date` with an incorrect format.

ENVIRONMENT VARIABLES

`TZ` If defined, the environment variable `TZ` contains information on timezones. `date` uses `TZ` to determine the timezone and to convert from UTC (Universal Time Coordinated) to local time and vice versa.

The value of `TZ` consists of

- the standard timezone
- the offset from UTC in hours
- if relevant, the summer timezone with information for conversions from standard time to summer time (daylight saving time) and back.

A detailed description of `TZ` is provided under `sh(1)` in the section "Standard shell variables".

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_TIME` environment variable governs the language and the format in which the date and time and the values of date and time constants are displayed.

`LC_CTYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES`, `LC_TIME` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1: Printing date and time (`LANG=En_US.ASCII`) [**Format 1**]

If you call `date` without arguments at 7:43 hours MET on March 27, 1997, and the system clock is set to the correct time, the following will be printed:

```
Thu Mar 27 07:43:10 MET 1997
```

The command:

```
$ date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

generates the following output:

```
DATE: 03/27/97
TIME: 07:59:51
```

Example 2: Setting the system clock [**Format 2**]

You are working under the system administrator login `root` and want to set the date and time to 16:30 on March 17, 2001. To do this, you must first switch to single-user mode and unmount all file systems. You then enter:

```
# date 0317163001
```

If you are calling `date` in the year 2001, there is no need to specify the year.

SEE ALSO

`cal`(1), `sysadm`(1M), `ctime`(3C), `printf`(3S), `LC_TIME`(4), `environ`(5).

User's Reference Manual

dc(1)

NAME

dc - desk calculator

SYNOPSIS

dc [*file*]

DESCRIPTION

dc provides you with the functions of a desk calculator for interactive use in Reliant UNIX.

file Name of the file containing the calculation operations to be executed. dc reads *file* to its end, and then reads from standard input.

file not specified:

dc reads from standard input.

MODE OF OPERATION OF DC

dc works on the stacking principle. A stack is a linear memory area that can only be accessed on a LIFO (Last In First Out) basis, i.e. the last value stored in the stack is located at the top, and is the only one accessible for operations. If values further down the stack need to be processed, the values stacked above them must either be deleted or temporarily stored in some other memory area. This memory area could be another stack or a register.

A register is an area of rapid-access memory in the CPU.

The memory area with which dc operates is generally referred to as a "stack" in the following discussion. The term "main stack" is used only if this memory needs to be differentiated from other memory areas (other stacks or registers).

dc accepts input in reverse Polish notation (postfix notation). Thus, to add 3 and 4, you would enter:

```
3 4 +
```

dc operates on decimal integers by default. You may define another base for input and output (see **SYNOPSIS OF DC PROGRAMS**, i or o), however, and change the number of fractional digits (see **SYNOPSIS OF DC PROGRAMS**, k).

dc is called by bc (see bc). bc is a preprocessor for dc; it allows you to enter operators and operands in normal arithmetic notation and provides a C-like syntax that implements functions. Control structures can also be included in bc programs.

SYNOPSIS OF DC PROGRAMS

c The entire contents of the stack are popped (deleted).

d The top value on the stack is duplicated (loaded onto the stack a second time).

f All values on the stack are printed, but retained in the stack.

i The top value on the stack is used as the base for further input; the value itself is popped from the stack. The default value for the base is 10, i.e. inputs are interpreted with base 10.

I The current value of the input base is pushed onto the stack.

k The top value on the stack is popped and used as a non-negative scale factor. In other words, a corresponding number of decimal places are printed for subsequent output and maintained during multiplication, division and exponentiation.

l*x* The contents of register *x* are pushed onto the stack. Register *x* remains unchanged. The initial value of all registers is 0.

L*x* The top value is popped, i.e. deleted, from stack *x* and pushed onto the main stack.

o The top value on the stack is popped and used as the base for further output. Numbers are output to

the base 10 by default.

`O` The currently valid base for output is pushed onto the top of the stack.

`p` The top value on the stack is printed but not removed from the stack.

`P` The top value on the stack is interpreted as an ASCII string and printed. In this case, the value is removed from the stack.

`q` Terminates the program. If a loop is being executed, the recursion level is popped by two.

`Q` Terminates the program. The top value is popped from the stack and the execution level of the loop is popped by that value.

`s`*x* The top value of the stack is popped and loaded into a register named *x*, i.e. stored as the top element in register *x*. *x* may be any character.

`S`*x* The top value of the main stack is popped and pushed onto a stack named *x*, i.e. stored as the top element in stack *x*. *x* may be any character.

`v` The top value on the stack is replaced by its square root, taking all the fractional part into account (even if the applicable scale factor is less than the number of existing fractional digits, see `k`).

`x` The top value on the stack is interpreted as a series of statements to the `dc` command and executed.

`X` The last loaded value is replaced by the currently valid number of fractional digits.

`z` The stack level (the number of elements it contains is pushed onto the stack.

`Z` The top number on the stack is replaced by its length.

number

The value of *number* is pushed onto the stack. *number* is a string comprising the digits 0-9 and periods and may be preceded by an underscore. The first period is interpreted as a decimal point; all others are ignored.

A preceding underscore is interpreted as a negative sign.

`!` The remainder of the line from the exclamation point onward is interpreted as a Reliant UNIX command.

`+`
`–`
`/`
`*`
`%`
`^`

The top two values on the stack have one of the specified operations applied to them. The meanings are:

`+` addition

`–` subtraction

`/` division

`*` multiplication

`%` remaindering

`^` exponentiation

Both entries are popped from the stack, and the result of the executed operation is pushed onto the stack instead.

`;:` `bc` uses the characters `;` and `:` for array operations.

`<`*x*
`>`*x*
`=`*x*

The top two elements of the stack are popped and compared. If the specified relation is satisfied, i.e. the topmost value is respectively less than, greater than, or equal to the second value, register *x* is evaluated.

`?` A line of input is taken from the input device (usually the display terminal) and executed.

`[...]`
The bracketed ASCII string is placed on top of the stack.

ERROR MESSAGES

x `is unimplemented`
x is an octal number that is not a valid statement for `dc`.

`stack empty`
Not enough elements on the stack to do what was asked.

`Out of space`
No more space in the stack (too many digits).

`Out of headers`
Too many numbers.

`Out of pushdown`
Too many items on the stack.

`Nesting Depth`
Too many levels of nested execution.

`exp not an integer`
The exponent is not an integer.

EXAMPLES

Example 1

In this example, the numbers 0-9 will be printed:

```
[lip1+sili10>a]sa
0si lax
0
1
2
3
4
5
6
7
8
9
```

`dc` processes this entry as follows: The string `lip1+sili10>a` is first pushed on the top of the stack ([]) and then popped from it and stored in register `a` (`sa`).

`0si` causes the number 0 to be stored in register *i*. The string `lip1+sili10>a` is then pushed onto the stack, but remains stored in register *a* as well. The command *x* causes the string `lip1+sili10>a` to be interpreted as a series of `dc` commands and to be evaluated accordingly.

The resulting sequence of operations is as follows: the number 0 (contents of register `i`) is pushed on the stack and printed, etc. *Example 2*

This example prints the first ten values of `n!`:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
1
2
6
24
120
720
5040
```

```
40320
362880
3628800
```

SEE ALSO

bc(1) .

User's Reference Manual

dd(1)

NAME

`dd` - convert and copy files

SYNOPSIS

`dd` [*option ...*]

DESCRIPTION

The `dd` command can be used to copy files while making any necessary conversions, e.g. from EBCDIC to ASCII or vice versa.

`dd` is well-suited for input/output via raw devices (raw physical I/O), because it can read and write in blocks of any size. Note, however, that any given raw device can only handle the block sizes for which it is specifically designed (512-byte blocks for a floppy disk drive, for example). If the size of the file you want to convert is not a multiple of the appropriate block size, you must use the `conv=sync` option.

After completion, `dd` reports the number of whole and partial input and output blocks on standard error.

Caution:

Never use `dd` to copy files between file systems which have different block sizes.

Using a block device to copy a file may result in extra nulls being added to the file to pad the final block to the block boundary.

OPTIONS

Option	Brief description
`if=`*input_file*	Name of input file
`of=`*output_file*	Name of output file
`ibs=`*n*	Input block size (in bytes)
`obs=`*n*	Output block size (in bytes)
`bs=`*n*	Input and output block size (in bytes)
`cbs=`*n*	Conversion block size (in bytes)
`files=`*n*	Number of files to be copied and concatenated
`skip=`*n*	Skip first *n* blocks of input file
`iseek=`*n*	Seek *n* blocks from beginning of input file before copying
`oseek=`*n*	Seek *n*+1 blocks from beginning of output file before copying
`seek=`*n*	Seek *n*+1 blocks from beginning of output file before copying
`count=`*n*	Copy/convert only first *n* blocks of input file
`conv=...,...`	Several values for `conv`
`conv=ascii`	Convert from EBCDIC to ASCII
`conv=ebcdic`	Convert from ASCII to EBCDIC
`conv=ibm`	Convert from ASCII to IBM-specific EBCDIC
`conv=block`	Convert newline-terminated records to fixed-length records
`conv=unblock`	Convert fixed-length records to newline-terminated records
`conv=lcase`	Convert uppercase letters to lowercase

conv=ucase	Convert lowercase letters to uppercase
conv=swab	Swap every pair of bytes
conv=noerror	Do not stop processing on error
conv=sync	Pad every input block to the number of characters defined in ibs=*n*
conv=notrunc	Do not truncate the output file

No option specified:
dd reads from standard input and writes to standard output. No conversions are performed. The input and output block sizes default to 512 bytes.

If size arguments are needed for an option, you can specify them as follows:

n means *n* times 1

*n*b means *n* times 512

*n*k means *n* times 1024

*n*w means *n* times 2

*n*x*m* means *n* times *m*

*n***m* means *n* times *m*

if=*input_file*
In *input_file* you give the name of the input file from which dd is to read.
if=*input_file* not specified:
dd reads from standard input.

of=*output_file*
output_file designates the name of the output file to which dd is to write.
of=*output_file* not specified:
dd writes to standard output.

ibs=*n* (input block size) In *n* you specify the input block size in bytes.
ibs=*n* not specified:
The input block size defaults to 512 bytes.

obs=*n* (output block size) In *n* you specify the output block size in bytes.
obs=*n* not specified:
The output block size defaults to 512 bytes.

bs=*n* (block size) In *n* you specify a value in bytes for both the input and output block size. The value specified for bs overrides the values of ibs and obs.
bs=*n* not specified:
The input block size is preserved instead of packing short blocks into the output buffer (this is particularly efficient since no in-core copy need be done).

cbs=*n* (conversion block size) Specify the conversion block size for block and unblock in bytes by *n* (default is zero). If cbs=*n* is omitted or given a value of zero, using block or unblock produces unspecified results.
This operand must also be specified if the conv= operand is specified with a value of ascii, ebcdic or ibm. For a conv= operand with an ascii value, the input is handled as described for the unblock value except that characters are converted to ASCII before any trailing space characters are deleted. For conv= operands with ebcdic or ibm values, the input is handled as described for the block value except that the characters are converted to EBCDIC or IBM EBCDIC, respectively, after any trailing space characters are added.

(see `conv=ascii`, `conv=ebcdic`, `conv=ibm`, `conv=block` and `conv=unblock`).

`files=`*n*
: In *n* you specify the number of files `dd` is to copy and concatenate before terminating. This option only makes sense when input is from a magnetic tape unit or a similar device (such as a streamer).

`skip=`*n*
: The first *n* blocks of the input file are skipped, and copying begins at block number *n*+1. This option is only appropriate for magnetic tape, where `iseek` is undefined.

`iseek=`*n*
: The first *n* blocks of the input file are skipped, and copying begins at block number *n*+1. This option is specially designed for disk files, where `skip` can be extremely slow. It is only accepted with devices which support the `lseek` system call (such as disk units).

  `skip` and `iseek` are mutually exclusive.

`oseek=`*n*
: The copy of the input file (after conversion, if specified) begins *n* blocks into the output file (i.e. at block *n*+1). The first *n* blocks of the output file remain unchanged. This options is only accepted with devices which support the `lseek` system call (such as disk units).

`seek=`*n*
: `seek` performs the same function as `oseek`.

`count=`*n*
: Only the first *n* blocks of the input file are copied (and converted, if specified). If *n* is 0, the complete file is copied.

`conv=...,...`
: You can specify a number of values for `conv`, separated by commas, and in this way bring about a number of conversions.

  The values `ascii`, `ebcdic`, and `ibm` cannot be specified simultaneously. The same applies for the value pairs `block` and `unblock`, as well as `lcase` and `ucase`.

`conv=ascii`
: Conversion from EBCDIC to ASCII.

  The number of characters defined in `cbs=`*n* is copied into the conversion buffer and any specified character mapping is done. Trailing blanks are removed, and a newline character is added. The output block size of the file is thus adjusted to the value defined in `cbs=`*n*. If `cbs` is unspecified or set to 0, input file character mapping is done, but trailing blanks are not removed.

`conv=ebcdic`
: Conversion from ASCII to EBCDIC.

  The ASCII characters are read into the conversion buffer and converted to EBCDIC. Trailing blanks are added in order to produce the output block size defined in `cbs=`*n*. If `cbs` is unspecified or set to 0, input file character mapping is done, but the file's block structure is not changed.

`conv=ibm`
: Conversion from ASCII to EBCDIC as described above, but using an IBM-specific EBCDIC table.

`conv=block`
: Conversion of newline-terminated ASCII records to fixed-length records. The ASCII characters are read into the conversion buffer, and blanks are added to adjust the record length or output block size to the value defined in `cbs=`*n*. If `cbs` is unspecified or set to 0, this option produces a simple file copy.

`conv=unblock`
: Conversion of fixed-length ASCII records to newline-terminated records. The record length is determined by the value defined in `cbs=`*n*. This number of characters is repeatedly read into the conversion buffer, where excess trailing blanks are removed and a newline character is added before the characters are sent to the output. If `cbs` is unspecified or set to 0, this option produces a simple file copy.

`conv=lcase`

Uppercase letters are converted to the corresponding lowercase letters.

`conv=ucase`
: Lowercase letters are converted to the corresponding uppercase letters.

`conv=swab`
: The bytes in each pair of bytes are swapped. If an input block contains an uneven number of bytes, the last byte is ignored.

`conv=noerror`
: Processing is continued even if an error occurs.

`conv=sync`
: Every input block is padded to the value specified in `ibs=`*n*. When `block` or `unblock` is specified, blanks or null-bytes are used.

`conv=notrunc`
: The output file is not regenerated. All blocks that are not overwritten explicitly by `dd` remain unchanged.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The contents of an EBCDIC tape with a block structure of 10 times 80 bytes is to be written to the ASCII file `xy`:

```
$ dd if=/dev/rmt32 of=xy ibs=800 cbs=80 conv=ascii
```

SEE ALSO

`cp(1)`.

User's Reference Manual

deroff(1)

NAME

deroff - remove nroff, troff, tbl and eqn constructs

SYNOPSIS

deroff [*option ...*] [*file ...*]

DESCRIPTION

deroff removes all nroff and troff requests, backslash constructs, macro calls, eqn constructs (between .EQ and .EN lines, and between delimiters), and tbl table descriptions from a file or from standard input and replaces them with white space. It then writes the remainder of its input to standard output.

deroff follows chains of included files (.so and .nx commands); if a file has already been included, a .so naming it is ignored and a .nx naming it terminates execution.

OPTIONS

No option specified:

deroff filters out the above formatting constructs from its input and writes the remainder to standard output.

−i (ignore) forces deroff to ignore all .so and .nx commands.

−m[m|l|s]

(m − macros) The standard macro packages mm and ms are included.

−mm causes the mm macros to be interpreted such that only running text is output [i.e. no text (parameter) from macro lines]. −ms functions analogously.

−ml automatically sets −mm as well and also causes lists associated with the mm macros to be deleted.

−w The output is a word list, one "word" per line, with all other characters deleted. Otherwise, the output follows the original, with the changes mentioned above.

In text, deroff sees a "word" as any string that *includes* at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a "word" is a string that *begins* with at least two letters and includes a total of at least three letters.

Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from "words".

file Name of the file to be filtered. More than one file may be named.

file not specified:

deroff reads from standard input.

EXAMPLES

The text of the troff file text.der is displayed with cat in unfiltered form and then filtered with deroff:

```
$ cat text.der
'\"macro stdmacro
.if n .pH g1.basename @(#)basename      40.5 of 10/10/89
.\" Copyright
.nr X
.if \nX=0 .ds x} basename 1 "User Environment Utilities" "\&"
.if \nX=1 .ds x} basename 1 "User Environment Utilities"
.if \nX=2 .ds x} basename 1 "" "\&"
.if \nX=3 .ds x} basename "" "" "\&"
.TH \*(x}
.if t .ds ' \h@.05m@\s+4\v@.333m@\'\v@-.333m@\s-4\h@.05m@
$ deroff text.der
```

```
macro stdmacro
Copyright
     basename  User Environment Utilities
     basename  User Environment Utilities
     basename
     basename
```

SEE ALSO

eqn(1M), mm(1M), mmt(1M), neqn(1M), nroff(1M), tbl(1M), troff(1M), man(5), me(5), mm(5), ms(5), mv(5).

deroff(1-ucb)
BSD Compatibility Package

NAME

`deroff` - remove `nroff`, `troff`, `tbl` and `eqn` constructs

SYNOPSIS

`/usr/ucb/deroff` [`–w`] *filename* ...

DESCRIPTION

The `deroff` command reads each file in sequence and removes all `nroff` and `troff` command lines, backslash constructions, macro definitions, `eqn` constructs (between `.EQ` and `.EN` lines or between delimiters), and table descriptions and writes the remainder on the standard output. `deroff` follows chains of included files (`.so` and `.nx` commands); if a file has already been included, a `.so` is ignored and a `.nx` terminates execution. If no input file is given, `deroff` reads from the standard input file.

OPTIONS

`–k` if `deroff` is called with the `–ms` or `–me` option (see below): causes keep blocks associated with the `ms` macros [`.KS`/`.KF`/`.KE` and `.DS`/`.DE`] and keep blocks and lists associated with the `me` macros [`.(z`/`.)z` and `.(e`/`.)e`] to be preserved.

`–m`[`a`|`e`|`m`|`s`|`l`]
the standard macro packages `man`, `me`, `mm`, and `ms` are included.

`–ma` causes the `man` macros to be interpreted such that only running text is output [i.e. no text (parameter) from macro lines]. `–me`, `–mm`, and `–ms` functions analogously.

`–ml` causes keep blocks and lists associated with the `mm`/`ms` macros to be deleted.

`–w` Generate a word list, one word per line. A "word" is a string of letters, digits, and apostrophes, beginning with a letter; apostrophes are removed. All other characters are ignored.

NOTES

`deroff` is not a complete `troff` interpreter, so it can be confused by subtle constructs. Most errors result in too much rather than too little output.

`deroff` does not work well with files that use `.so` to source in the standard macro package files.

SEE ALSO

`eqn`(1M), `mm`(1M), `mmt`(1M), `neqn`(1M), `nroff`(1M), `tbl`(1M), `troff`(1M), `man`(5), `me`(5), `mm`(5), `ms`(5), `mv`(5).

User's Reference Manual

destroy(1)

NAME

`destroy -` physically delete files

SYNOPSIS

`destroy` [*option ...*] *file ...*

DESCRIPTION

`destroy` overwrites files with selectable characters and then deletes the directory entry for each file. `destroy` can only be used to overwrite and delete regular files.

The `rm` command does not physically delete files from the disk; so it is theoretically still possible to read data processed by `rm` by using the block and raw device special files.

OPTIONS

No option specified:

The input files are overwritten with binary zeros and then deleted. `destroy` does not overwrite and delete files with multiple hard links (the link counter must be on 1).

If the file to be destroyed is a symbolic link, it is deleted from the directory and the file it references is overwritten with binary zeros.

`-c`*X* (character) The input files are overwritten with the character you specify here, instead of with binary zeros.

`-i` (interactive mode) `destroy` asks you for confirmation before deleting each file.

`-l` (linked files) `destroy` will also overwrite and delete files with multiple hard links (link counter set to more than 1). The file itself is destroyed and all links to it are removed.

`-l` not specified:

`destroy` does not overwrite and delete files with multiple links.

`-s` (silent mode) Error messages are suppressed.

file Name of the file you wish to destroy. `destroy` will only delete regular files. You may name more than one file.

ERROR MESSAGES

`destroy: cannot overwrite` *filename* `(Permission denied)`

As you do not have write permission for *filename*, `destroy` cannot overwrite it.

`destroy: cannot destroy directory` *dirname*

You have tried to apply `destroy` to a directory *dirname*; `destroy` can only be used to delete regular files.

`destroy: file` *filename* `has` *n* `links -- not destroyed`

There are *n* links (*n* > 1) to the input file *filename*; `destroy` does not delete files with multiple links unless you use the `-l` option.

`destroy: cannot stat` *filename* `(No such file or directory)`

The file *filename* does not exist.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized. Answers to yes/no queries must be given in the language appropriate to the current locale.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

The current directory contains a file named `file`. You first use `ln` to make another link to this file:

```
$ ln file copy
$ ls –l
-rw-------   2 hugo   project   14  Jan 16 15:04 file
-rw-------   2 hugo   project   14  Jan 16 15:04 copy
$ cat file
Good morning!
$ cat copy
Good morning!
```

The file can now be accessed under both names, although it is physically present only once.

```
$ destroy file
destroy: file file has 2 links –– not destroyed
```

`destroy` does not overwrite and delete the file in this case, since it has two links.

```
$ destroy –l –cX file
```

`destroy` overwrites the file this time, since you have used the `–l` option. The `–cX` option causes the file contents to be overwritten with `X`. `destroy` then deletes the link named `file`.

```
$ ls –l
-rw-------   1 hugo   project   14  Jan 16 15:04 copy
$ cat copy
XXXXXXXXXXXXXX
```

SEE ALSO

`ln(1)`, `rm(1)`, `wcheck(1)`, `secure(1M)`.

df(1)

NAME

`df` - report number of free disk blocks and files (disk free)

SYNOPSIS

`df` [*option ...*] [*file ...*]

DESCRIPTION

`df` can be used to display how many free disk blocks and inodes are available on mounted or unmounted local file systems and on file systems mounted via NFS.

OPTIONS

The following options can be usefully combined with one another:

`–b`, `–e`, `–n`

`–k`, `–P`

All other options should not be used in combination since doing so makes them ineffective.

No option specified:
`df` prints the number of free blocks and inodes in all mounted or all specified file systems. Disk space is given out in 512 byte blocks.

`–F` *FSType*
FSType is the type of file system on which to operate. This option is only needed if the file system is unmounted.

FSType can be any of the following:

`hs` High Sierra file system

`fdfs` Pseudo file system for administering file descriptors

`nfs` File system mounted from a remote system (Network File System)

`proc` Pseudo file system for accessing process data

`ufs` UNIX System V Release 4 file system

`vxfs` Veritas file system (default)

`–b` Prints the free disk space (in Kbytes).

`–e` Prints the number of free inodes.

`–g` Prints the entire `statvfs` structure for all mounted file systems or for *FSType* or *file*. The output consists of 4 lines containing the following information:

dir Name of the directory on which the file system is mounted (the string *dir* does not appear in the output)

device
Associated special file (the string *device* does not appear in the output)

`block size`
Block size of the file system in bytes

`frag size`
Fragment size of the file system in bytes

`total blocks`
Total number of blocks in units of 512 bytes

`free blocks`
Total number of free blocks in units of 512 bytes

`available`

Number of free blocks for non-privileged users in units of 512 bytes

`total files`
Total number of inodes

`free files`
Number of free inodes

`filesys id`
File system identification number

name File system designation (the string *name* does not appear in the output)

`fstype`
File system type

`flag` File system flags

`filename length`
Maximum length of file names

`-k` Prints the following information in columns for all mounted file systems or for the specified file systems:

`filesystem`
Special file associated with the file system

`kbytes`
Total disk space, in Kbytes

`used` Used disk space, in Kbytes

`avail`
Available disk space, in Kbytes

`capacity`
Percentage of disk space used

`mounted on`
Name of the directory on which the file system is mounted

Note: The `avail` figure allows for the space available only for the nonprivileged user.

`-l` `df` prints the number of free blocks and inodes for all locally mounted file systems or for the specified file systems. The `-l` option is set by default.

`-n` `df` prints the file system type name.

`-P` `df` generates a portable format that can be incorporated into automatic procedures (e.g. shell scripts). The following information is printed in columns for all mounted file systems or for the specified file systems:

`Filesystem`
Special file associated with the file system

`512-blocks` or
`1024-blocks`
Total disk space

`Used` Used disk space

`Available`
Free disk space

`Capacity`
Percentage of disk space used

`Mounted on`
Name of the directory on which the file system is mounted

Note: The `Available` figure allows for the space available only for the nonprivileged user.

`-t` `df` prints the number of free blocks and files as well as the total number of available blocks and files for each mounted or specified file system.

–V `df` expands the `df` command line and writes the completed command to standard output, but does not execute the command. `df` completes the command line, adding information derived from `/etc/mnttab` and `/etc/vfstab` to input provided by the user.

The added information relates to the option –F *FSType* and/or the associated special file, depending on which of the two components, if either, was specified on the command line.

If you enter `df –V` without other options and arguments, you will be shown a list of `df` command lines indicating all mounted file system types and the associated special filenames.

–v `df` prints the following information in columns for all mounted file systems or for the specified file systems:

`Mount Dir`
: Name of the directory on which the file system is mounted

`Filesystem`
: Special file associated with the file system

`blocks`
: Total disk space, in 512-byte blocks

`used` Used disk space, in 512-byte blocks

`free` Free disk space, in 512-byte blocks

`%used`
: Percentage of disk space used

–o *spec_option*

–o cannot be combined with any of the generic command options.

There are specific options for the file system types `ufs` and `vxfs`:

–o *ufs_option*

If the file system type is `ufs`, you can use a `ufs`-specific option:

`i` `df` prints the following information in columns:

`Filesystem`
: Name of the associated special file

`iused`
: Number of used inodes

`ifree`
: Number of free inodes

`%iused`
: Percentage of inodes used

`Mounted on`
: Name of the directory on which the file system is mounted

–o *vxfs_option*

If the file system type is `vxfs`, you can use a `vxfs`-specific option:

`s` Outputs the number of free memory areas of each size. The lengths of free memory areas are always specified in exponents of 2. The minimum length is one block, and the maximum length cannot be greater than one assignment unit.

–– If *file* begins with a dash (–), the end of the command-line options must be marked with ––.

file *file* is either the name of a file or a directory, or the name of a special file containing a file system. When you specify a file or a directory, `df` refers to the file system in which the file or directory is contained. You can specify a number of names.

file not specified:

`df` examines all locally and remotely mounted file systems or all file systems of type *FSType*.

ERROR MESSAGES

`ufs usage: df [–F ufs] [generic_options] [–o i] [directory|special]`
`ufs usage: df [–F vxfs] [generic_options] [–o s] [directory|special]`

An invalid argument was specified for *ufs_option* or *vxfs_option* with the `–o` option. *ufs_option* can only be `i`, *vxfs_option* can only be `s`.

`df: operation not applicable for FStype` *xxx*

An inapplicable file system type (*xxx*) was specified for *FStype* with `df –F` *FStype* `–o i|s`. The only types that can be specified with the `–o` option are `ufs` or `vxfs`.

`df: cannot stat` *file*

Non-existent or invalid *file* (special file or directory).

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Printing the associated file system types for all user directories:

```
$ df –n /h*
/home    : ufs
/home2   : ufs
/home3   : ufs
```

Example 2

The entire `statvfs` structure is to be output for the `/stand` file system:

```
$ df –g /stand
/              (/dev/root             ):  16384 block size    2048 frag size
 139356 total blocks     83736 free blocks     69800 available
  16384 total files      12812 free files    1048576 filesys id
    ufs fstype      0x00000004 flag              255 filename length
```

FILES

`/dev/ios`*n*`/*`
: Special files associated with the file systems

`/etc/mnttab`
: Table of mounted file systems

`/etc/vfstab`
: List of default parameters for each file system

`/etc/fs/`*FSType*`/*`
: Commands for specific file system types

SEE ALSO

`df`(1-ucb), `df`(1-ufs), `df`(1-vxfs), `du`(1), `statvfs`(2).

Specific online call for a `df` manual page e.g. "man 1-ufs df".

User's Reference Manual

df(1-ufs)

NAME

`df(ufs)` - report free disk space on ufs file systems

SYNOPSIS

`df [ –F ufs ] [` *generic_options* `] [ –o i ] [` *directory* | *special ...* `]`

DESCRIPTION

The `df` command reports the number of free disk blocks and inodes for a `ufs` file system.

Note that the amount of space reported as used and available is less than the amount of space in the file system; this is because the system reserves a fraction of the space in the file system to allow its file system allocation routines to work well. The amount reserved is typically about 10%; this may be adjusted using `tunefs(1M)` . When all the space on the file system except for this reserve is in use, only the super-user can allocate new files and data blocks to existing files. When the file system is overallocated in this way, `df` may report that the file system is more than 100% utilized.

OPTIONS

`–F ufs`
: Specifies the `ufs`-FSType.

generic_options
: Options supported by the generic `df` command.

`–o` Specify `ufs` file system specific options. The available option is:

`i` Report the number of used and free inodes. May not be used with the options `–b`, `–e`, `–k`, `–t`, `–v`, `–g`, `–l`, `–n` of the generic `df` command.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

FILES

`/dev/ios*/*disk*`
`/etc/mnttab`
: list of file systems currently mounted

SEE ALSO

`df(1)` , `du(1)` , `quot(1M)` , `tunefs(1M)` , `mnttab(4)` .

User's Reference Manual

df(1-vxfs)
VxFS

NAME

`df(vxfs)` - report number of free disk blocks and inodes for aVeritas File System (VxFS)

SYNOPSIS

`df` [`–F vxfs`] [*generic_options*] [`–o s`] [*directory* | *special* ...]

DESCRIPTION

This manual page is an addendum to the `df(1)` manual page. It describes the Veritas File System (VxFS) specific features of the `df` command.

The `df` command prints out the number of free blocks and free inodes in Veritas file systems or directories by examining the counts kept in the super blocks. The *special* device name (for example, `/dev/ios0/sdisk001s3`) or mount point *directory* name (for example, `/usr`) must be specified. If *directory* is a directory name, the report presents information for the device that contains the directory.

OPTIONS

`–F vxfs`
: Specifies the Veritas file system type (*FSType*).

generic_options
: Options supported by the generic `df` command.

`–o`
: Specify Veritas file system specific options. The following option is available:

  `s`
  : Print the number of free extents of each size. Free extents are always an integral power of 2 in length, ranging from a minimum of 1 block to a maximum of the size of an allocation unit.

FILES

`/dev/ios*/*disk*`
`/etc/mnttab`
: mount table

`/etc/vfstab`
: list of default parameters for each file system

SEE ALSO

`df(1)`, `fs(4-vxfs)`.

User's Reference Manual

df(1-ucb)
BSD Compatibility Package

NAME

`df` - report free disk space on file systems

SYNOPSIS

`/usr/ucb/df` [`–i`] [`–a`] [`–t` *FSType* | *file* | *special* ...]

DESCRIPTION

`df` displays the amount of disk space occupied by currently mounted file systems, the amount of used and available space, and how much of the file system's total capacity has been used.

If arguments to `df` are path names, `df` produces a report on the file system containing the named file. Thus "`df .`" shows the amount of space on the file system containing the current directory.

OPTIONS

`–i` Report the number of inodes that are used and free.

`–a` Report allocation in kilobytes for all file systems including `/proc` and `/dev/fd`.

`–t` *FSType*
Specify type of file system with *FSType* (for example, `ufs`).

EXAMPLE

A sample of output for `df` looks like this:

```
/usr/ucb/df –a
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/root              49914   34292   10630    76%    /
/proc                      0       0       0     0%    /proc
/dev/fd                    0       0       0     0%    /dev/fd
/dev/ios0/sdisk000s2  331650  218966   79518    73%    /opt
/dev/ios0/sdisk000s3  552456  257504  239706    52%    /usr
/dev/ios0/sdisk000s4  188892   89712   80290    53%    /var
```

FILES

`/etc/mnttab`
List of file systems currently mounted

`/etc/vfstab`
List of default parameters for each file system

SEE ALSO

`du`(1), `quot`(1M), `tunefs`(1M), `mnttab`(4).

User's Reference Manual

dfspace(1)

NAME

`dfspace` - report number of free disk blocks

SYNOPSIS

`/usr/sbin/dfspace` [*file*]...

DESCRIPTION

`dfspace` allows you to check the number of free blocks from the total number available on the local file system. It also shows you this figure as a percentage.

ARGUMENTS

–– If *file* begins with a dash (–), the end of the command-line options must be marked with ––.

file *file* is either the name of a file or a directory, or the name of a special file containing a file system. When you specify a file or a directory, `dfspace` refers to the file system in which the file or directory is contained. You can specify a number of names.

file not specified:

`df` examines all local file systems.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Input:

```
dfspace
```

Output:

```
                    /   55.91 MB of   96.53 MB available ( 57.92%).
                 /proc    0.00 MB of    0.00 MB available (100.00%).
               /dev/fd    0.00 MB of    0.00 MB available (100.00%).
                  /usr   32.06 MB of  200.00 MB available ( 16.03%).
                  /var  174.69 MB of  200.00 MB available ( 87.35%).
                  /opt  237.48 MB of  300.00 MB available ( 79.16%).
                 /home  902.05 MB of 1099.99 MB available ( 82.00%).
            Filesystem   Disk Space
Total local Disk Space 1402.21 MB of 1896.53 MB available ( 73.94%).
```

Example 2

Input:

```
dfspace /home /opt
```

Output:

```
                 /home  902.05 MB of 1099.99 MB available ( 82.00%).
                  /opt  237.48 MB of  300.00 MB available ( 79.16%).
            Filesystem   Disk Space
Total local Disk Space 1139.53 MB of 1399.99 MB available ( 81.40%).
```

FILES

`/dev/iosn/*`
: Special files associated with the file systems

`/etc/mnttab`
: Table of mounted file systems

`/etc/vfstab`
: List of default parameters for each file system

SEE ALSO

`df`(1).

User's Reference Manual

diff(1)

NAME

`diff -` differential file comparator

SYNOPSIS

`diff` [*option ...*] *file1 file2*

DESCRIPTION

The `diff` utility will compare the contents of *file1* and *file2* and write to standard output a list of changes necessary to convert *file1* into *file2*. This list should be minimal. No output will be produced if the files are identical.

OPTIONS

No option specified:

If the compared files are identical, `diff` produces no output. When the files differ, the output is as follows:

1. line[range] from *file1* `ed` command line[range] from *file2*
2. lines that are only in *file1*
3. lines that are only in *file2*

The lines are displayed in the following format:

1. *n1*[,*n2*] `a` / `d` / `c` *n1*[,*n2*]

2. < text of line from *file1* □
. □
. □
. □
– – –
3. > text of line from *file2* □
. □
. □
.

`a`, `d`, and `c` are `ed`-like commands meaning:

`a` append

`d` delete

`c` change

The output is to be interpreted as follows:

The `ed` commands `a`, `d`, and `c` with their preceding line (or range) specifications show how *file1* can be converted into *file2*.

If you replace `a` with `d` and `d` with `a` and use the line (or range) specifications to the right, the commands indicate how *file2* can be converted into *file1*. Lines from *file1* are marked <; those from *file2* are marked >.

`–a` This option produces a diff with all lines of context from both files. Those lines only in *file1* are

prepended with –; those only in *file2* are prepended with +. Lines which are identical in both files are prepended with .

–b `diff` ignores trailing blanks or tabs at the end of lines as well as differing numbers of blanks at corresponding positions within text lines. Leading blanks and blank lines are reported as differences.

–i `–i` must not be combined with `–h`.

`diff` ignores the case of letters; for example, it considers "A" to be the same as "a".

–t `diff` expands tab characters in output lines. In normal or `–c` output, `diff` adds character(s) to the start of each line that may adversely affect the indentation of the original source lines and make the output lines difficult to interpret. The `–t` option preserves the original source's indentation.

–w `diff` ignores all blanks (space and tab characters) and treats strings of blanks of any length as equivalent; for example, it considers "if (a = = b)" to be the same as "if(a==b)".

The following options are mutually exclusive:

–c `diff` produces a listing in three parts with a slightly modified output format. The output begins with the names and creation dates of *file1* and *file2*. Then come the lines that differ, with the lines not present in *file2* marked with a minus sign (–) and the lines that differ in *file1* and *file2* marked with an exclamation point (!). There are also three lines of context before and after the differing lines.

–C *number*

Produces a listing of differences identical to that produced by `–c`, but with *number* lines of context before and after each difference.

–e The `–e` option must not be combined with `–h`, `–l` or `–s`.

`diff` produces output suitable for use as an `ed` script. The `ed` script contains `a`, `d`, and `c` commands and the related text lines as input for the editor `ed` to convert *file1* to *file2*. Before you pass the script to the editor, you should add the statements `w` and `q` at the end of the script and insert the command `e` *file1* at the start [see ed(1)].

–f The `–f` option must not be combined with option `–h`.

`diff` generates a similar script to the one created with option `–e`, but converting in the opposite direction. This script, however, is **not** usable with `ed`.

Editing scripts produced under the `–e` and `–f` options may be incorrect when dealing with lines comprising nothing but a single period.

The following options modify the way in which `diff` works:

–h The `–h` option must not be combined with `–e`, `–f`, `–i`, `–c`, `–C`, `–n` or `–D`.

`diff` operates faster, and you can process files of any length. However, the result produced by option `–h` is not reliable!

–n Produces a script similar to `–e`, but with the `ed` commands listed in reverse order. In addition, a count of the lines to be changed is printed after each `insert` or `delete` command.

–D *string*

In this case *file1* and *file2* must be C source programs or contain C source fragments.

`diff` creates a merged version of *file1* and *file2* with C preprocessor controls included so that compilation of the result is equivalent to compiling *file1* if *string* is not defined, and is equivalent to compiling *file2* if *string* is defined.

The following options are used to compare directories:

–l `diff` produces output in long format. Before checking for differences, `diff` pipes each text file through `pr` to paginate it. Differences other than those in text files are remembered and are summarized after all text file differences have been reported.

–r Applies `diff` recursively to all common subdirectories encountered.

–s Reports files that are identical; these would not otherwise be mentioned.

–S *name*
: Starts a directory `diff` in the middle, beginning with the file *name*.

––
: If *file1* begins with a dash (–), the end of the command line options must be marked with ––.

file1 file2
: Names of the files `diff` is to compare. If *file1* is a directory, then a file in that directory with the same name as *file2* is compared against *file2*. If *file2* is a directory, then the comparison is made with a file named *file1* from that directory.

EXIT STATUS

`0` Files are the same

`1` Files are different

`>1` Input error

ERROR MESSAGES

`diff: two filename arguments required`

You have specified an incorrect number of files. Only two files can be compared at a time.

`diff: No such file or directory`

One of the specified files does not exist.

`diff: files too big, try –h`

You need to use the `–h` option, as the files to be compared are too large.

`diff: Permission denied`

You do not have read permission for one of the specified files.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The files `file1` and `file2` have the following contents:

```
file1 file2
Jack Jill            Jack and Jill□
went up the hill     went uphill□
to fetch a pail      of water□
of water
```

By calling `diff`, you can identify the lines in which the two files differ.

```
$ diff file1 file2□
1,3c1,2□
<Jack Jill□
<went up the hill□
<to fetch a pail□
–––□
>Jack and Jill□
>went uphill
```

Explanation: In order to convert `file1` into `file2`, lines 1 to 3 inclusive from `file1` (1,3) must be replaced (c) by lines 1 to 2 inclusive (1,2) of `file2`. The contents of each of these lines are shown in the output lines

beginning with the < or > characters. Lines in `file1` are marked <; lines in `file2` are marked >.

Example 2

Compare files and produce an `ed` script:

Contents of `file1`: Contents of `file2`:

```
today is Monday          today is Tuesday
it is cold               it is autumn
                          it is cold
```

After the following command, `diff` outputs the `ed` commands with which `ed` could convert `file1` into `file2`. To be able to use this output as input for `ed`, you need to add the statements `w` and `q` to the end of the `ed` script.

```
$ diff –e file1 file2
1c
today is Tuesday
it is autumn
 .
$
```

SEE ALSO

bdiff(1), cmp(1), comm(1), diff3(1), dircmp(1), ed(1), pr(1).

User's Reference Manual

diff3(1)

NAME

`diff3 -` three-way differential file comparator

SYNOPSIS

`diff3` [*option ...*] *file1 file2 file3*

DESCRIPTION

`diff3` compares three files, outputs the ranges of lines in which they differ to standard output, and indicates what changes need to be made to make them identical.

OPTIONS

No option specified:

In the case of line ranges that do not match, `diff3` flags the ranges of text with the following codes:

`====` all 3 files differ

`====1` *file1* differs from *file2* and *file3*

`====2` *file2* differs from *file1* and *file3*

`====3` *file3* differs from *file1* and *file2*

This flag is followed by the changes that have to be made in order to convert a given range in one file to a given range in some other file. Such changes are output in one of the following two formats:

Change text:

f`:`*n1*[`,`*n2*]`c`

[*text*]

To eliminate the differences, line *n1* or the range *n1* to *n2* must be changed (c) in *file f* (where *f* = 1, 2, or 3). *text* is the original contents of the line range to be changed. If the contents of two files do not differ in the given range, *text* will not be displayed for the file with the lower *f* number.

Append text:

f`:`*n*`a`

[*text*]

To eliminate the differences, text must be appended (a) after line *n* in *file f* (where *f* = 1, 2, or 3). *text* is the line range to be appended. If the contents of two files do not differ in the given range, *text* will not be displayed for the file with the lower *f* number.

OPTIONS

`–e` `diff3` produces an `ed` script which will incorporate into *file1* all changes between *file2* and *file3*, i.e. the changes that would be flagged `====` or `====3` if no option were specified [see `ed`(1)].

Caution:

The `–e` option is ignored in the case of text lines that consist of nothing but a single period (.).

Examples:

Contents of

file1	*file2*	*file3*
`maybe`	`surely`	`maybe`
`tomorrow`	`today`	`today`

Calling `diff3 –e` generates the following output:

```
$ diff3 –e file1 file2 file3
```

```
1,2c
maybe
today
 .
w
q
```

Contents of

file1	*file2*	*file3*
maybe	maybe	maybe
		today
		tomorrow

Calling `diff3 –e` generates the following output:

```
$ diff3 –e file1 file2 file3
1a
today
tomorrow
 .
w
q
```

`–E` `diff3` produces an `ed` script which will incorporate into *file1* all changes between *file2* and *file3*, i.e. the changes that would be flagged `====` or `====3` if no option were specified [see ed(1)].

In this case, however, `diff3` will expand the output in the `ed` script for changes that would be flagged `====` if no option were specified. These changes are enclosed within `<<<<<<` or `>>>>>>` lines, depending on the file where they are located.

`–x` `diff3` produces an `ed` script which will incorporate into *file1* only those changes between *file2* and *file3* that would be flagged `====` if no option were specified [see ed(1)].

`–X` `diff3` produces an expanded `ed` script which will incorporate into *file1* only those changes between *file2* and *file3* that would be flagged `====` if no option were specified [see ed(1)].

These changes are enclosed within `<<<<<<` or `>>>>>>` lines, depending on the file where they are located.

`–3` `diff3` produces an `ed` script which will incorporate into *file1* only those changes between *file2* and *file3* that would be flagged `====3` if no option were specified [see ed(1)].

file1 file2 file3

Names of the three files that you want to compare.

`diff3` will find a maximum of 200 differences between the three files.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

You have three versions of an address file, called `adr1`, `adr2` and `adr3`. The contents of these files are shown below:

Contents of `adr1`: Contents of `adr2`: Contents of `adr3`:

```
Mervyn Parry      Mervyn Parry      Mervyn Parry
3 Kings Rd.       3 Kings Rd.       3 Kings Rd.
London W5         London W5         London W5
Tel.: 01-987654   Tel.: 01-987654   Tel.: 01-987654
Saul Mahoney      Saul Mahoney      Saul Mahoney
83 Charles St     83 Charles St     83 Charles St
Baltimore, USA    Baltimore, USA    Baltimore, USA
Tel.: 747 1510    Tel.: 747 1450    Tel.: 747 1450
                                    Robert Jeffery
                                    4 Grosvenor Rd
                                    Acton W3
                                    01-777888
```

Example 1

Calling `diff3` without options:

```
$ diff3 adr1 adr2 adr3
====
1:9c
  Tel.: 747 1510
2:9c
  Tel.: 747 1450
3:9,14c
  Tel.: 747 1450
  Robert Jeffery
  4 Grosvenor Rd
  Acton W3
  01-777888
```

Example 2

Calling `diff3` with option `-e`:

```
$ diff3 -e adr1 adr2 adr3
9c
Tel.: 747 1450
Robert Jeffery
4 Grosvenor Rd
Acton W3
01-777888
 .
w
q
```

If you redirect the output of `diff3 -e` to a file named `script`, you can incorporate the changes in `adr1` with the following command line:

```
$ (cat script; echo '1,$p') | ed - adr1
```

Example 3

Calling `diff3` with option `-3`:

```
$ diff3 -3 adr1 adr2 adr3
```

This call does not generate any output, since `adr1` and `adr2` are not identical. `diff3 -3` produces an `ed` script only when there are changes of type `====3`; all differences in this example are of the `====` type.

FILES

`/tmp/d3a*`
: File created by `diff` containing the differences between *file1* and *file3*. `diff` is called by `diff3`.

`/tmp/d3b*`
: File created by `diff` containing the differences between *file2* and *file3*.

`/usr/lib/diff3prog`
: Program called by `diff3` to compare the files `/tmp/d3a*` and `/tmp/d3b*`.

SEE ALSO

`diff`(1), `ed`(1).

User's Reference Manual

dircmp(1)

NAME

`dircmp -` directory comparison

SYNOPSIS

`dircmp` [`–d`] [`–s`] [`–w` *n*] [`––`] *dir1 dir2*

DESCRIPTION

`dircmp` compares two directories *dir1* and *dir2* and tells you whether they contain files of the same name and whether such files also have the same contents.

OPTIONS

No option specified:

`dircmp` first lists the file names that are unique to each directory. Files from *dir1* are shown on the left, while files from *dir2* appear on the right. The output is generated in the following format:

`Nov 16 10:42 1987` *dir1* `only` *dir2* `only Page 1`

./filename1 ./filename2
..
..
..
..

The next set of lines provides information as to whether files common to the two directories have the same contents. If this is the case, they are listed in a line that begins with `same`; if not, in a line that begins with `different`. The generated output is in the following format:

`directory .`

`different` *./filename in dir1 and dir2*
.
.
.
`same` *./filename in dir1 and dir2*

`–d` `dircmp` compares the contents of identically named files in *dir1* and *dir2* and outputs the results in a `diff`-like format. The result of this comparison is output immediately after the file name comparison.

`–s` `dircmp` suppresses information about the contents of common files.

`–w` *n* The width of the output line is changed to *n* characters. The default width is 72 characters per line.

`––` If *dir1* begins with a dash (–), the end of the command-line options must be marked with ––.

dir1 dir2

Names of the directories you wish to compare.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Comparison of the two directories `A` and `B`, which have the following contents:

Directory	File names	File contents
A	a	aaaaaaa
	b	bbbb
	c	cccc
	d	dddd
B	a	aaaaaaaaa
	b	BBBB
	C	cccc
	d	dddd

Calling `dircmp` with the `–d` option:

```
$ dircmp –d A B
Nov  16  10:42  1987  A only and B only Page 1
 ./c                   ./C
Nov  16  10:42  1987  Comparison of A B Page 1
directory .
different ./a
different ./b
same      ./d
Nov  16  10:42  1987  difference of ./a in A and B Page 1
1c1
< aaaaaaa
---
> aaaaaaaaa
Nov  16  10:42  1987  difference of ./b in A and B Page 1
1c1
< bbbb
---
> BBBB
```

SEE ALSO

`cmp(1)`, `diff(1)`.

User's Reference Manual

dirname(1)

NAME

`dirname` - strip file name from full access path

SYNOPSIS

`dirname` [--] *string*

DESCRIPTION

`dirname` can eb used to strip the file name (basename) from the full access path. `dirname` takes a string as its argument, removes the final slash and all characters to the right of it, and writes the rest of the string on standard output. `dirname` is useful in shell scripts.

OPERANDS

-- If *string* begins with a dash (-), the end of the command line options must be marked with --.

string

string can be any string of characters.

`dirname` deletes the final slash (and any slashes directly in front of it) and all characters to the right of it and writes the remaining portion to standard output. If string does not contain a slash, a dot is written to standard output.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

In the following example, the path prefix `/usr/src/cmd` is assigned as a value to the `NAME` variable:

```
NAME=`dirname /usr/src/cmd/xyz.c`
```

SEE ALSO

`basename`(1), `ed`(1), `sh`(1).

User's Reference Manual

doscat(1)

NAME

`doscat -` display files on an MS-DOS floppy disk

SYNOPSIS

`doscat [ –m ] [ –r ] a:`*dospath ...*

DESCRIPTION

`doscat` writes one or more files from an MS-DOS floppy disk to standard output on the Reliant UNIX system.

OPTIONS

No option specified:
The code for line feed in MS-DOS text files is CR-NL; Reliant UNIX files use NL only. The sequence CR-NL is converted to NL. The DOS end-of-file character SUB ("1A") is eliminated. An MS-DOS text file is converted into a Reliant UNIX text file. No character conversion is performed on binary files.

`–m` Forces character conversion, which means that the character conversions described above are also performed for binary files.

`–r` `doscat` transfers data "raw", i.e. all characters remain unchanged (including those in text files).

`a:`
`A:`
As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dospath
specifies the path name of the required file in the MS-DOS file system.

The use of wild cards (*, ?, [ ]) is permitted but the path name must be unique.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

What are the contents of the MS-DOS file "BOOKS"?

```
$ doscat a:books
This file lists all books purchased by the company.
Author          Title                           Purchase        Price
                                                 date$
```

Example 2

You want to read the contents of the MS-DOS files "USER/JONES/ORDERS" and "GAMES/FISH".

```
$ doscat a:user/jones/orders a:games/fish
ORDERS since 01.01.1986
FISH-GAME:
If this screen were an aquarium,
one could have a
little fish swimming around in it!
```

```
$
```

The `doscat` command has displayed the contents of the files sequentially.

NOTES

If a directory contains more than 254 file names, the `doscat` command may produce the error message "`Error – too many filenames`". In this case, the user must limit the number of file names involved (for example by not using "*.*").

User's Reference Manual

doscp(1)

NAME

`doscp -` copy files between Reliant UNIX and MS-DOS

SYNOPSIS

Format 1

`doscp [-m][-r][-l]` *unixfile* `a:`*dospath*

Format 2

`doscp [-m][-r][-l]` *unixfile ...* `a:`*dosdirectory*

Format 3

`doscp [-m][-r][-l] a:`*dosfile unixpath*

Format 4

`doscp [-m][-r][-l] a:`*dosfile ... unixdirectory*

DESCRIPTION

`doscp` copies one or more Reliant UNIX files to an MS-DOS floppy disk, or one or more MS-DOS files from an MS-DOS floppy to Reliant UNIX.

A distinction is made between the following four formats of the `doscp` command:

- copy a Reliant UNIX file to MS-DOS (**Format 1**)
- copy several Reliant UNIX files to an MS-DOS directory (**Format 2**)
- copy an MS-DOS file to Reliant UNIX (**Format 3**)
- copy several MS-DOS files to a Reliant UNIX directory (**Format 4**)

OPTIONS

No option specified:

The code for line feed in MS-DOS text files is CR-NL; Reliant UNIX files use NL only. When copying from MS-DOS to Reliant UNIX, the sequence CR-NL is converted to NL. The DOS end-of-file character SUB ("1A") is eliminated.

When files are copied from Reliant UNIX to MS-DOS, NL is converted to CR-NL, and the DOS end-of-file character SUB ("1A") is added.

An MS-DOS text file is converted into a Reliant UNIX text file. No character conversion is performed on binary files.

Note:

To ensure that your requirements regarding character conversion are met it is advisable to use the `-m` or `-r` option as appropriate.

`-m` Forces character conversion, which means that the character conversions described above are also performed for binary files.

`-r` `doscp` transfers data "raw", i.e. all characters remain unchanged (including those in text files).

`-l` When copying from Reliant UNIX to MS-DOS, the file names are converted to uppercase automatically. When copying files from MS-DOS to Reliant UNIX, the names remain the same, i.e. in uppercase.

The `-l` option can be used to convert all uppercase letters in file names to lowercase when copying from MS-DOS to Reliant UNIX.

unixfile

Name of the Reliant UNIX file.

`a:`
`A:`

As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dospath

Path name of the MS-DOS directory on the MS-DOS floppy disk, indicating the directory to which the Reliant UNIX file is to be copied. The use of wild cards (*, ?, [ ]) is permitted but the path name must be unique.

If there is already an MS-DOS file with the same name as the Reliant UNIX file, *dospath* can also be used to give the Reliant UNIX file a different name on the MS-DOS system.

dospath not specified:

`doscp` copies the Reliant UNIX file to the root directory on the MS-DOS floppy.

dosdirectory

Name of the MS-DOS directory to which the Reliant UNIX file or files is/are to be copied. The use of wild cards (*, ?, [ ]) is permitted but the directory specified must be unique.

dosfile

Name or path name of the MS-DOS file on the MS-DOS floppy. The use of wild cards (*, ?, [ ]) is permitted.

unixpath

Name of the Reliant UNIX directory or file.

unixdirectory

Name of the Reliant UNIX directory.

Caution:

To ensure that you data can be read on a DOS-based system, special conventions must be observed when selecting MS-DOS files names. In MS-DOS, file names may consist of a basename comprising a maximum of 8 characters, optionally followed by a period and a file name extension (i.e. suffix) of at most 3 characters. If excess characters are specified for the basename and the extension, they are truncated to 8 and 3 characters respectively. Some extensions have special meanings in DOS. Unlike Reliant UNIX, MS-DOS does not allow the use of all special characters. Appropriate error messages are issued in the event of errors. DOS does not distinguish between uppercase and lowercase letters in file names. Thus, if you specify names in lowercase, they will be converted to uppercase on the disk and will appear in uppercase the next time the floppy is read.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

You want to copy the Reliant UNIX file `schedule` to the directory `user/jones` on an MS-DOS floppy disk.

```
$ doscp schedule a:user/jones
```

Example 2

You want to copy the Reliant UNIX file `schedule` to an MS-DOS floppy disk and simultaneously rename it `april`.

```
$ doscp schedule a:april
```

Example 3

You want to copy all Reliant UNIX files that have names beginning with `max` to an MS-DOS floppy disk.

```
$ doscp max* a:
```

You can use the dosdir(1) or dosls(1) command to verify that the files are actually on the MS-DOS floppy.

```
$ dosdir a:
MAX      1        712   1-20-86   5:09p
MAX      2       8075   1-20-86   5:09p
MAX      3      18727   1-20-86   5:09p
MAX      4       3417   1-20-86   5:10p
```

Example 4

You want to copy the MS-DOS file BOOKS to the Reliant UNIX directory /usr/guest and rename it library.

```
$ doscp a:BOOKS library
```

Example 5

You want to copy the files GAMES/WORM and USER/APRIL from the MS-DOS floppy to your current Reliant UNIX directory.

```
$ doscp a:games/worm a:user/april .
```

Example 6

You want to copy the MS-DOS files SCHULTZ/CHARLIE and SCHULTZ/SNOOPY and the files GRIMM/HANSEL and GRIMM/GRETEL to the Reliant UNIX directory usr/john/tales.

```
$ doscp a:schultz/charlie a:schultz/snoopy a:grimm/hansel a:grimm/gretel
/usr/john/tales
```

NOTES

If a directory contains more than 254 file names, the doscp command may produce the error message "Error – too many filenames". In this case, the user must limit the number of file names involved (for example by not using "*.*").

SEE ALSO

dosdir(1), dosls(1).

User's Reference Manual

dosdir(1)

NAME

`dosdir -` list directories on an MS-DOS floppy disk in MS-DOS format

SYNOPSIS

`dosdir a:`*dospath ...*

DESCRIPTION

`dosdir` produces a DOS-format listing of the contents of one or more directories on an MS-DOS floppy disk, plus the amount of free space available on the floppy.

`a:`
`A:`
: As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dospath
: specifies the path name of the required directory on the MS-DOS floppy disk.

The use of wild cards (*, ?, [ ]) is permitted.

dospath not specified:

`dosdir` lists the contents of the root directory on the MS-DOS floppy.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

List the contents of the root directory on an MS-DOS floppy disk:

```
$ dosdir a:
GAMES        <DIR>   530  1-15-86   2:58p
BIN          <DIR>  1292  1-15-86   2:58p
USER         <DIR>   478  1-15-86   2:58p
COUNTERS     <DIR>  2249  1-15-86   2:58p
PROGRAM      <DIR>  8463  1-15-86   2:58p
       5 File(s)    161472 Bytes free
```

Example 2

List the contents of a subdirectory on an MS-DOS floppy:

```
$ dosdir a:games
.            <DIR>     728  1-15-86   2:58p
..           <DIR>    5153  1-15-86   2:58p
FISH               1   438  1-15-86   3:09p
YAHTZEE            1   917  1-15-86   3:09p
COUNTERS           1  2384  1-15-86   3:09p
SPIRALS            1  6725  1-15-86   3:09p
WORM               1   839  1-15-86   3:09p
ENTERPR  MSD       1  1277  1-15-86   3:20p
MASTERM  MSD       1  1082  1-15-86   3:20p
```

```
    9 File(s)         78278 Bytes free
```

Example 3

List the contents of a number of directories on an MS-DOS floppy:

```
$ dosdir a:games a:user
/GAMES:
     .            <DIR>         ...
     ..           <DIR>         ...
     FISH               1       ...
     YAHTZEE            1       ...
     COUNTERS           1       ...
     SPIRALS            1       ...
     WORM               1       ...
     ENTERPR  MSD       1       ...
     MASTERM  MSD       1       ...
/USER:
     .            <DIR>         ...
     ..           <DIR>         ...
     JONES        <DIR>         ...
```

User's Reference Manual

dosfilt(1)

NAME

`dosfilt` - make files with language-dependent special charactersreadable under MS-DOS

SYNOPSIS

`dosfilt` *–char_set* [–o *outfile*] [*file ...*]

DESCRIPTION

`dosfilt` converts Reliant UNIX files containing language-dependent special characters in 7-bit format (such as ä, ö, ü, ß) or data in the ISO 8859-1 8-bit character set [see `ascii(5)` and `meta-ascii(5)`] so that they can read and edited under MS-DOS (IBM character set). NL is converted to CR-NL, and the DOS end-of-file character SUB ("1A") is added. By default `dosfilt` writes to standard output.

char_set

Language or character set (in the Reliant UNIX file) to be converted to the IBM character set.

You can select any of the following languages or character sets:

`–a`	Danish
`–b`	Belgian
`–d`	German
`–e`	Spanish
`–f`	French
`–g`	British
`–i`	Italian
`–n`	Norwegian
`–s`	Swedish
`–z`	Swiss
`–x`	This option can be used to convert a file containing ISO 8859-1 characters to a file containing characters from the IBM character set [see `meta-ascii(5)`].

`–o` *outfile*

Name of the output file.

file Name of the file to be converted.

file not specified: `dosfilt` reads from standard input.

To convert MS-DOS files to Reliant UNIX files you use the `sinfilt(1)` command.

The commands `sinfilt –x` and `dosfilt –x` convert the specified files from the IBM character set to the ISO 8859-1 character set (`sinfilt`) and vice-versa (`dosfilt`). Since these two character sets do not contain exactly the same range of characters, the following restrictions should be noted:

- 52 characters (including all ASCII characters and German umlauts) can be converted directly on a 1 : 1 basis.
- Some country-specific special characters in the ISO 8859-1 character set (44 characters in all) cannot be directly converted to identical characters in the IBM character set, where they do not appear in the same form. These characters are replaced by similar characters.
- All semigraphic characters and a large number of math and Greek characters in the IBM character set (a total of 63 characters with hex codes above 7f) cannot be converted directly to identical characters from the ISO character set. They are replaced by similar characters.

EXAMPLES

Converting the Reliant UNIX file `schedule` in the ISO 8859-1 character set to the MS-DOS file `SCHEDULE` in the IBM character set:

```
$ dosfilt –x –o SCHEDULE schedule
```

SEE ALSO

`sinfilt`(1), `ascii`(5), `meta-ascii`(5).

User's Reference Manual

dosformat(1)

NAME

`dosformat` - format an MS-DOS floppy disk

SYNOPSIS

`dosformat` [*option ...*] *special_file*

DESCRIPTION

`dosformat` formats a floppy disk for computers running the MS-DOS operating system. The various types of formats are described below.

OPTIONS

No option specified:

`dosformat` formats the floppy disk in the format defined by *special_file*.

`-f` Suppresses the query as to whether there is a floppy disk in the drive and begins formatting immediately.

`-q` Suppresses information normally displayed on the screen during the formatting process. The query as to whether there is a floppy in the drive is not suppressed.

`-v` `dosformat` prompts for a volume label to identify the floppy disk more precisely. A volume label can have a maximum of 11 characters. Volume labels are displayed by the `dosdir(1)` command.

special_file

Here you must specify one of the special file names listed in the table below. This file determines the format of the floppy disk.

The following floppy disk formats (3½" only) can be generated:

Tracks	Sectors	Gross storage capacity in KB	Net storage capacity in KB	Special file name to be specified when formatting
80	9	720	713	`[r]sfdisk???f3dt` `[r]fx3dt`
80	18	1440	1423.5	`[r]sfdisk???f3ht` `[r]fx3ht`
80	36	2880	2846	`[r]sfdisk???f3et`
				`[r]sfdisk???ft` `(dynamic floppy recognition)`

Caution:

The pathnames of device files vary depending on Reliant UNIX versions:

Reliant UNIX RM600

`/dev/ios?`, e.g. `/dev/ios?/[r]sfdisk???f3dt`

Reliant UNIX RM400

`/dev/at/flp`, e.g. `/dev/at/flp/[r]fx3dt`

When formatting with the autodetection device, the format depends on the physical properties of the floppy disk.

You cannot format, read, or write to single-sided floppy disks.

The net storage capacity indicates the storage area of the floppy disk that can actually be used. The remaining capacity contains administrative information about the floppy. As files are always stored on a floppy in blocks of 512 bytes or 1 KB or 2 KB (depending on the format), the net storage capacity can lie considerably below those values shown in the table if a large number of small files are copied.

Floppy disks formatted in this way can be processed with the DOS commands on a Reliant UNIX computer and can be used on MS-DOS computers.

Caution:

The `dosformat` command does not check whether there is already data on the floppy disk. Any files already stored there will be overwritten when the floppy is formatted.

ED floppy disks can be formatted on ED, HD and DD. HD floppy disks can be formatted on HD and DD. DD floppy disks can only be formatted on DD. If the autodetector device is used, the floppy disk is formatted according to its physical format.

Note:

MS DOS cannot read wrongly formatted floppy disks. It is therefore possible, for example, that an ED floppy disk formatted on HD cannot be recognized (because of missing floppy disk indicators).

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The following command formats a 3½" floppy disk in the first disk drive to 80 tracks and 18 sectors per track. The formatted storage capacity of the floppy disk will be 1440 KB.

`dosformat /dev/ios0/rsfdisk000f3ht` (for an RM600)

`dosformat /dev/at/flp/rfx3ht` (for an RM400)

SEE ALSO

`dosdir(1)`, `gettypes(1)`.

User's Reference Manual

doslabel(1)

NAME

`doslabel` - label an MS-DOS diskette

SYNOPSIS

`doslabel {a:|A:}`

DESCRIPTION

`doslabel` writes a label, which is requested interactively, on a diskette.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

```
$ doslabel a:
Volume label (11 characters, <RETURN> for none)? BACKUP  001
$ dosdir a:
Volume in drive A: is BACKUP  001
Directory of A:
         0 File(s)   1457664 Bytes free
```

SEE ALSO

`dosdir(1)`, `dosformat(1)`.

User's Reference Manual

dosls(1)

NAME

`dosls` - list directories on an MS-DOS floppy disk in Reliant UNIXformat

SYNOPSIS

`dosls a:`*dospath ...*

DESCRIPTION

`dosls` produces a Reliant UNIX format listing of the contents of one or more directories on an MS-DOS floppy disk.

`a:`
`A:`
: As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dospath
: Specifies the path name of the required directory on the MS-DOS floppy disk. The use of wild cards (*, ?, [ ]) is permitted.

  dospath not specified:

  `dosls` lists the contents of the root directory on the MS-DOS floppy.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

List the contents of the root directory on an MS-DOS floppy:

```
$ dosls a:
GAMES
BIN
USER
ACCOUNTS
PROGRAMS
```

Example 2

List the contents of a subdirectory on an MS-DOS floppy:

```
$ dosls a:user
JONES
```

Example 3

List the contents of a number of directories on an MS-DOS floppy:

```
$ dosls a:user a:games
/GAMES:
     FISH
     YAHTZEE
     COUNTERS
     SPIRALS
```

```
      WORM
      ENTERPR.MSD
      MASTERM.MSD
/USER:
      JONES
```

NOTES

If a directory contains more than 254 file names, the `dosls` command may produce the error message "`Error – too many filenames`". In this case, the user must limit the number of file names involved (for example by not using "*.*").

It is also sufficient with the `dosls` command to simply specify the path name in order to avoid the error message.

User's Reference Manual

dosmkdir(1)

NAME

`dosmkdir` - create directories on an MS-DOS floppy disk

SYNOPSIS

`dosmkdir a:`*dospath* ...

DESCRIPTION

`dosmkdir` creates one or more directories on an MS-DOS floppy disk.

`a:`
`A:`
: As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dospath
: Specifies the path name of the new directory in the MS-DOS file system.

  To ensure that data in MS-DOS files can be read on a DOS-based system, special conventions must be observed when choosing names for MS-DOS directories. Directory names with a basic name of more than 8 characters and a suffix of more than 3 characters are truncated accordingly (i.e. to 8 and 3 characters, respectively). In other words, the name of an MS-DOS directory may consist of a basename of up to

  8 characters and an optional suffix of up to 3 characters, with a period between the basename and the suffix. Unlike Reliant UNIX, MS-DOS does not allow the use of all special characters. Appropriate error messages are issued in the event of errors. Furthermore, DOS does not distinguish between uppercase and lowercase letters. Thus, if you specify names in lowercase, for example, they will be converted to uppercase on the floppy disk, and will appear in uppercase letters the next time the floppy is read.

`dosmkdir` generates the following links for each directory:

"." for the directory itself, and

".." for its parent directory.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The directory "address" is to be created:

```
$ dosmkdir a:address
$ dosls a:
GAMES
BIN
USER
ACCOUNTS
PROGRAMS
ADDRESS
```

`dosls` shows that the `dosmkdir` command has been executed.

Example 2

Create a number of directories:

Further subdirectories are now to be set up under the directory "ADDRESS" created in Example 1. These directories are to be named "MUNICH" and BAVARIA". In addition, the directory "ORDERS" is to be created in the directory "USER/MAIER".

Enter the following command:

```
$ dosmkdir a:address/munich a:address/bavaria a:user/maier/orders
$ dosdir a:address a:user/maier
/ADDRESS:
     .            <DIR>
     ..           <DIR>
     MUNICH       <DIR>
     BAVARIA      <DIR>
/USER/MAIER:
     .            <DIR>
     ..           <DIR>
     ORDERS       <DIR>
```

`dosdir` shows that the `dosmkdir` command has been executed, and that the new directories have been created.

User's Reference Manual

dosrm(1)

NAME

`dosrm` - delete MS-DOS files

SYNOPSIS

`dosrm a:`*dosfile ...*

DESCRIPTION

`dosrm` deletes one or more MS-DOS files from an MS-DOS floppy disk. Full path names must be specified for the files to be deleted. Note that there is no protection against the unauthorized deletion of a file, and deleted files cannot be retrieved.

`a:`
`A:`
: As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dosfile
: Name or path name of the MS-DOS file in the MS-DOS file system. The use of wild cards (*, ?, [ ]) is permitted.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

You want to delete the MS-DOS file "HORSES".

```
$ dosrm a:horses
```

Example 2

You want to delete the MS-DOS file with the path name "BODY/HEART" from the MS-DOS floppy disk.

```
$ dosrm a:body/heart
```

Example 3

You want to delete the MS-DOS files "MAIL", "TELLER" and "TOKEN" from the MS-DOS floppy disk.

```
$ dosrm a:mail a:teller a:token
```

Example 4

You want to delete the MS-DOS files with the path names "FLOWERS/CLIVIA", "ANIMAL/MAMMAL/HEDGEHOG", and "STONES/GRANITE".

```
$ dosrm a:flowers/clivia a:animal/mammal/hedgehog a:stones/granite
```

NOTES

If a directory contains more than 254 file names, the `dosrm` command may produce the error message "`Error – too many filenames`". In this case, the user must limit the number of file names involved (for example by not using "*.*").

User's Reference Manual

dosrmdir(1)

NAME

`dosrmdir -` delete directories from an MS-DOS floppy disk

SYNOPSIS

`dosrmdir a:`*dosdir ...*

DESCRIPTION

`dosrmdir` deletes one or more directories from the MS-DOS file system on an MS-DOS floppy disk. Directories must be empty before they can be deleted.

`a:`
`A:`
: As in DOS conventions, `a:` or `A:` designates the disk drive via which the MS-DOS floppy disk is addressed. This string must be prepended to each DOS path name. `a:` or `A:` addresses the first floppy disk drive. If a second drive exists, it may be addressed as `b:` or `B:`.

dosdir
: Path name of the directory to be deleted in the MS-DOS file system. The use of wild cards (*, ?, [ ]) is permitted.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Delete a single directory.

The directory "USER" on your MS-DOS floppy disk contains the following subdirectories:

```
$ dosls a:user
.
..
JONES
KHAN
```

You want to delete the directory "USER/KHAN":

```
$ dosrmdir a:user/khan
```

Example 2

Delete a number of directories.

The directory "USER" contains the entries:

```
$ dosdir a:user
.            <DIR>   476   1-16-86  10:43a
..           <DIR>   263   1-16-86  10:43a
JONES        <DIR>  2745   1-16-86  10:43a
KHAN         <DIR>   834   1-16-86  11:18a
LOBER        <DIR>  1476   1-16-86  11:19a
       5 File(s)    ... Bytes free
```

The directory "ADDRESS/OTHER" contains the entries:

```
$ dosdir a:address/other
```

```
.            <DIR>   273   1-16-86  11:17a
..           <DIR>   185   1-16-86  11:17a
ZIP1         <DIR>   947   1-16-86  12:39p
ZIP7         <DIR>   204   1-16-86  12:39p
       4 File(s)    ... Bytes free
```

You want to delete the subdirectory "LOBER" from the directory "USER", and the subdirectory "ZIP1" from the directory "ADDRESS/OTHER". After deletion you use the command dosls(1) to display the contents of both directories again.

```
$ dosrmdir a:user/lober a:address/other/zip1
$ dosls a:user a:address/other
/USER:
     .
     ..
     JONES
     KHAN
/ADDRESS/OTHER:
     .
     ..
     ZIP7
```

dosls(1) shows that the directories have been deleted.

SEE ALSO

dosdir(1), dosls(1).

User's Reference Manual

dot(1)

NAME

`dot, . -` execute shell scripts in current shell

SYNOPSIS

`.` *file*

DESCRIPTION

The shell built-in `.` (dot) executes the specified shell script in the current shell.

If you define new shell variables in the shell script or modify the values of existing shell variables, these variables take effect in the current shell environment.

If you unset shell options in the shell script with the shell built-in `set`, these options are set/unset in the current shell.

When you call the shell script, you can pass keyword parameters to it, but you cannot redefine the positional parameters. However, within the shell script you can access the positional parameters of the current shell.

If you use `set` to redefine the positional parameters within the shell script, the new values apply in the current shell.

ARGUMENTS

file Name of the shell script to be executed in the current shell. If *file* does not contain a slash, the shell searches for *file* in all directories whose path names have been assigned to the `PATH` variable.

Read permission is required for the specified file, but the file searched for need not be executable.

EXIT STATUS

Returns the value of the last command executed, or a zero exit status if no command is executed.

ENVIRONMENT VARIABLES

`PATH` Search path of the shell

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1:

When you create or modify your `$HOME/.profile` file, the commands and assignments contained in it are normally not executed until you next log in (open a login shell).

The dot command enables you to make these changes effective in the current shell itself, provided that the current directory is assigned to the `PATH` variable:

```
$ . .profile
```

Example 2:

```
$ cat foobar
foo=hello bar=world
$ . foobar
$ echo $foo $bar
hello world
```

SEE ALSO

ksh(1), set(1), sh(1).

User's Reference Manual

du(1)

NAME

du - summarize disk usage

SYNOPSIS

du [*option ...*] [*name ...*]

DESCRIPTION

du outputs the amount of disk space used by directories, subdirectories, and ordinary files in blocks of 512 bytes.

OPTIONS

No option specified:

If *name* is a directory, du lists the space occupied by the specified directory and all its subdirectories. The disk space occupied by ordinary files in the specified directory is included in the count, but not listed individually.

−a If *name* is a directory, du individually lists the space occupied by all the files in it and its subdirectories.

−k du outputs the amount of disk space in blocks of 1024 bytes.

−r If *name* designates a directory for which you do not have read permission, or a file that cannot be opened, du issues an error message.

−r is always set implicitly and for reasons of compatibility is only available as an option.

−s du displays only the grand total of disk space used by the named file or subtree.

−s must not be combined with −a.

−x If *file* is a directory, du only lists the space occupied by the subdirectories and files belonging to the same file system as *file*.

−− If *name* begins with a dash (−), the end of the command-line options must be marked with −−.

name Name of the file or directory for which disk usage is to be displayed. A file with two or more links is only counted once. A file that has a hole in it (e.g. if only block 1 and block 100 are used) will result in an incorrect block count.

name not specified:

The disk space occupied by the current directory and all its subdirectories is displayed.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

List the disk usage, in 512-byte blocks, of all subdirectories of the current directory whose names begin with DIR. The space occupied by ordinary files is included, but not listed individually.

```
$ du DIR*
6       DIR-1
136     DIR-2/SUB-1
```

```
140     DIR-2
5       DIR-3
54      DIR-4
52      DIR-5
```

Example 2

List the disk usage, in 512-byte blocks, of all subdirectories of the current directory whose names begin with `DIR`. The disk usage of ordinary files is listed individually in this case because the `–a` option is used.

```
$ du –a DIR*
1       DIR-1/file1
1       DIR-1/file2
1       DIR-1/file3
6       DIR-1
0       DIR-2/file4
1       DIR-2/file5
34      DIR-2/SUB-1/file6
99      DIR-2/SUB-1/file7
136     DIR-2/SUB-1
140     DIR-2
1       DIR-3/file8
1       DIR-3/file9
5       DIR-3
50      DIR-4/file10
1       DIR-4/file10.bak
54      DIR-4
50      DIR-5/file11
52      DIR-5
```

SEE ALSO

`df(1)`, `ls(1)`.

User's Reference Manual

du(1-ucb)
BSD Compatibility Package

NAME

`du` - display the number of disk blocks used per directory or file

SYNOPSIS

`/usr/ucb/du [ –F ufs ]`

`/usr/ucb/du [ –F ufs ] [ –a ] [ –s ] [ *filename* ... ]`

DESCRIPTION

`du` gives the number of kilobytes contained in all files and, recursively, directories within each specified directory or file *filename*. If *filename* is missing, "`.`" (the current directory) is used.

A file which has multiple links to it is only counted once.

OPTIONS

`–a` Generate an entry for each file.

`–s` Only display the grand total for each of the specified *filename*s.

Entries are generated only for each directory in the absence of options.

EXAMPLE

Here is an example of using `du` in a directory. We used the `pwd(1)` command to identify the directory, then used `du` to show the usage of all the subdirectories in that directory. The grand total for the directory is the last entry in the display:

```
% pwd
/usr/ralph/misc
% du
5       ./jokes
33      ./squash
44      ./tech.papers/lpr.document
217     ./tech.papers/new.manager
401     ./tech.papers
144     ./memos
80      ./letters
388     ./window
93      ./messages
15      ./useful.news
1211 .
%
```

NOTES

Filename arguments that are not directory names are ignored, unless you use `–a`.

If there are too many distinct linked files, `du` will count the excess files more than once.

SEE ALSO

`df(1)`, `pwd(1)`, `quota(1M)`.

dumpmsg(1)

NAME

dumpmsg - generate a message source file from a message catalog(dump message)

SYNOPSIS

dumpmsg [–C] *catfile msgfile*

DESCRIPTION

dumpmsg reads a binary coded message catalog from a file or from standard input and generates readable message source text from it. The dumped message text is written to a file or to standard output.

dumpmsg outputs only the messages, not the associated delimiting characters (e.g. double quotes).

OPTIONS

–C Characters not included in the ASCII character set are output as octal numbers in the form *xxx*.

catfile

Name of the file containing the X/Open-style binary coded message catalog.

If you use a dash (–) as the name for *catfile*, dumpmsg reads from standard input.

msgfile

Name of the file to which the message text is to be written.

If you use a dash (–) as the name for *msgfile*, dumpmsg writes to standard output.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The message source file hallo.msf is generated by calling extract. From this file the gencat command then generates the binary coded message catalog hallo.cat. Then dumpmsg is called to write the messages in the catalog in readable form to standard output (as indicated by the dash –):

```
$ cat hallo.c
main()
{
  printf("Hello world!");
}
$ extract hallo.c
...
$ gencat hallo.cat hallo.msf
$ dumpmsg hallo.cat –
$set 1
1 Hello world!
```

SEE ALSO

gencat(1), extract(1).

e(1) - fsirand(1-ucb)

e(1)

NAME

e - the Rand editor

SYNOPSIS

e [*options*] [*mainfile* [*altfile*]]

DESCRIPTION

The Rand editor e is a full screen editor which treats a text file as a two-dimensional surface with text on it. Among its more unusual features are manipulating rectangular areas of text, editing several files at once, and displaying text in one or more windows. While it is not as powerful as emacs, or as widely used as vi, most people at Rand prefer its easy to learn commands and use it to edit both programs and text. e is a complete rewrite of the Rand NED editor of several years back. The current version (E18) has been improved to take advantage of terminals that have the ability to do local insert/delete line, and it can now use terminfo so that most terminals with direct cursor addressing can be used.

The *mainfile* on the command line is the file which is initially displayed in the editor. If the file doesn't exist, the user will be asked if it should be created. If in addition, an *altfile* is specified on the command line, it will be read in as the initial alternate file. If no filenames are present on the command line, e will attempt to continue from your previous editing session. To do this it saves "state" in a file called .es1 in your current working directory. Before invoking the editor, be sure and specify your terminal type by setting the TERM environment variable.

OPTIONS

–help
: This causes a display of the currently available options and their values along with the version number of the editor.

–bullets or –nobullets
: Options –bullets and –nobullets forces border bullets on or off regardless of line speed or terminal type. Line speeds of 1200 baud and slower and some terminal types set border bullets off in the absence of these options.

–i[nplace]
: Option –inplace preserves file links except for singly-linked files. From within the editor, link preservation can be turned on and off, even for singly-linked files, by the update command.

–te[rminal]=*termtype*
: Option –terminal=*termtype* identifies the terminal type explicitly. In the absence of the –terminal option the terminal type is retrieved from the TERM environment variable.

  If there is no compiled-in code for the terminal, the terminfo entry for the terminal type will be used and the "standard" keyboard will be assigned.

–ke[yboard]=*termtype*
: Option –keyboard=*termtype* sets the keyboard type to *termtype*. It overrides any –terminal= argument and also any EKBD environment variable.

–kb[file]=*keyboardfile*
: Option –kbfile=*keyboardfile* replaces the keys assignment as described in *keyboardfile*. This option overrides any EKBFILE environment variable.

–notracks
: Option –notracks will allow you to edit a file without using or disturbing the work files (keys, state, and change files) from your previous session.

–l[ock]
: Option –lock enables the editor to lock the edited files and to perceive locked files by other sessions. This is not possible via NFS.

–reg[exp]
Option –regexp will allow you to state regular expressions for searching and replacing.

–replay=*filename*
Option –replay=*filename* allows you to replay with a specified keys file, where *filename* represents that keys file. Use –silent when the replay display is not desired. The message "***replay completed" or "***replay aborted" signals the end of the replay. The <CTRL-E> key can be used to stop the replay short of completion. Files .ec1* and .ek1* must not be present if the replay is to work.

–norecover
Option –norecover will open the edit session as defined by the .es1 state file, ignoring any recovery processing of a prior crashed or aborted session.

–v[iew]
Option –view allows you only to look at the files but not to change them.

COMMANDS

The following commands are available to the user by opening the command line via the <CMD> key and entering the command. The entered command is parsed and executed when the <RETURN> key is typed.

In the following discussion, brackets ("[", "]") indicate options, and vertical bar ("|") indicates optional choices.

There are two types of areas: a range of lines (indicated by *range*) and a marked rectangle of text (indicated by *rectangle*). Some commands will take either type of area, this is indicated by *area*.

A *range* can be specified either by marking or by a number followed by "l" for lines or "p" for paragraphs; rectangular areas must be marked. The default range is one line for most commands (fill, adjust and center default to one paragraph).

b | bye | ex | exit [n[o[save]]] | quit | a[bort] | dump]
Exit the editor. The different exit options affect whether or not the editor updates the edited files or not, and whether the ".e" state files are changed. The following table gives the complete exit story.

	Save files	Update state	Remove key file	Remove change fil
exit	X	X	X	
exit nosave	–	X	X	
exit quit	–	–	X	
exit abort	–	–	–	
exit dump	–	–	–	

```
blot     adjust | close | erase | pick | run | box
–blot    adjust | close | erase | pick | run | box
cover    adjust | close | erase | pick | run | box
insert   adjust | close | erase | pick | run | box
overlay  adjust | close | erase | pick | run | box
underlay adjust | close | erase | pick | run | box
```

All of these commands take the named buffer (one of adjust, close, erase, pick, run, or box) and add or merge it into the current file.

insert takes the named buffer, moves the existing text to make room for it, and then inserts it at the current text location. It is identical to the –pick command, except that now the other named buffers can also be inserted.

cover places the named buffer over top of the text without moving anything.

overlay is like cover, except that only the printing characters in the buffer clobber the original

text.

`underlay` is the reverse of `overlay`, i.e. the buffer covers only the non-printing characters of the original text.

`blot` is like `overlay` except that positions in the file corresponding to printing characters in the buffer are erased.

`–blot` is like `blot` except that positions in the file corresponding to blanks in the buffer are erased.

[g]box *rectangle*
Draws a box around the rectangle with "+", "|", and "–" characters.

ce[nter] [*range*] w[idth]=*n*
fi[ll] [*range*] w[idth]=*n*
ju[stify] [*range*] w[idth]=*n*
These commands all act upon the range of text and then replace them with the modified results. `center` centers the text around the column (width/2), `fill` places as many words as possible on each line, and `justify` is like `fill` except that it also provides a smooth right-margin by embedding blanks. The old text is saved in the adjust buffer.

cl[ose] [*area*]
–cl[ose]
`close` deletes the text from the file and remembers it in a close buffer, `–close` puts the close buffer back into the file at the current cursor position.

command
–command
Usually you issue commands one at a time by typing the <CMD> key, the command, and then <RETURN>. The `command` command places you in a mode where you can keep typing commands without hitting the <CMD> key each time. To return to normal edit mode, issue the `–command` command.

delete
Tells the editor to delete the current file on exit.

e[dit] [*filename*]
If no filename is specified, the alternate file (if any) is brought up in the current window. Otherwise the specified file is displayed.

er[ase] [*area*]
–erase
`erase` replaces the area by blanks and puts the contents into an erase buffer; `–erase` inserts the contents of this buffer at the current cursor position.

feed [*range*] *command-string*
A shell is started with the *command-string*, and the text in *range* is supplied to the shell as the standard input. Any results of the command are inserted before the first line in the range.

goto | go | g [b | *line-number* | e | rb | re]
`goto` is a quick way of moving around a file. `goto b` moves to the beginning of the file, `goto e` moves to the end. A `goto` *line-number* goes to that line of the file; `goto rb` and `goto re` move to the beginning or end of a marked range (see the `range` command).

jo[in] | –split
sp[lit] | –join
`join` (or `–split`) combines two lines, `split` (or `–join`) breaks a line in two.

n[ame] *newfilename*
Tells the editor to rename the current file to *newfilename* when it exits.

op[en] [*area*]
Move existing text to make room for *area* worth of blank lines (or a rectangle of blanks).

p[ick] [*area*]
–pi[ck]

`pick` copies the *area* to a pick buffer, `–pick` inserts the pick buffer at the current cursor position.

`range` [*range*]
`–range`
`?range`
: The `range` command is another method of limiting the range of some commands (notably the `replace`, `fill`, `justify` and `center` commands). `–range` turns it off, `?range` tells you the line numbers of the range area.

`red`[`raw`]
: The `redraw` command is used to erase and redraw the editor windows in case something has happened to it (for example, line noise on dialup lines, or messages from the operator).

`rep`[`lace`] [*range*] [*option*] /*search-string*/*replace-string*/
`–rep`[`lace`] [*range*] [*option*] /*search-string*/*replace-string*/
: The `replace` command searches forward over the range replacing the *search-string* with the *replace-string*; `–replace` searches backwards doing the same thing. The "/" delimiter can be replaced by other symbols if the search or replace strings contain a slash. Any non-control, non-alphanumeric character can be used as the delimiter.

  Two options can also be specified, they are `show` and `int`[`eractive`]. The `show` option allows the user to see the replacements as they occur, the `interactive` option additionally allows the user control over whether or not to make each replacement.

`run` [*range*] *command-string*
: The `run` command is similar to the `feed` command, except that the marked text is deleted and replaced by the results of running the *command-string*.

`sa`[`ve`] *newfilename*
: Immediately write a copy of the file to *newfilename*.

`tab` [*column ...*]
`–tab` [*column ...*]
: `tab` sets tabs in the specified columns; `–tab` removes tabs in specified columns.

`tabs` *n*
`–tabs`
: `tabs` sets tabs every *n*th column, and `–tabs` removes all tabs.

`tabfile` *tabfilename*
`–tabfile` *tabfilename*
: `tabfile` sets tabs (`–tabfile` clears tabs) at every column listed in the *tabfilename*.

`track`
`–track`
: The `track` command is used to scroll the main and alternate files together. This is normally used to visually compare the main file and the alternate file by rapidly changing between them and watching what changes on the screen (similar to blink-comparators as used in astronomy). The `–track` command turns off the tracking mode.

`up`[`date`] | `–up`[`date`] [`i`[`nplace`] | `–i`[`nplace`]]
: The `update` command tells the editor what to do on exit with the current file. The `update` command can be used to specify whether or not to break links, the `–update` command causes any changes to the current file to be ignored.

`w` | *window_cursor-pos* [*filename*]
`–w` | `–window`
: The `window` command will create a new window at the current cursor position (as long as the cursor is along the top or left margins). If a *filename* is specified, that file is displayed in the new window, otherwise the current file is used. The `–window` command deletes the current window.

STANDARD EDITOR KEYBOARD

There is now a "standard" `e` keyboard, that is designed to be usable on all video display terminals. Either say `e –keyboard=standard` or `setenv EKBD standard` before running e to select this keyboard. This is also the keyboard that e will use if there is no specific knowledge of your type of terminal compiled into the editor.

This keyboard layout is designed to be used on terminals with no function keys. `^H` means <CTRL-H> and `^X-^U` means <CTRL-X> followed by <CTRL-U>. The `ALT` entry gives you a choice of `^/` or `^_` because one or the other or both will not work on some terminals (_ is correct ASCII.)

Standard `e` keyboard

Key (in alphabetical order) =	Function	Function (in alphabetical order)	Key
^@	CMD	(extend)	^X
^A	CMD	+LINE	^F
^B	-WORD	+PAGE	^R
^C	BSP	+SRCH	^Y
^D	-LINE	+TAB	^I
^E	-PAGE	+WORD	^N
^F	+LINE	-LINE	^D
^G	HOME	-PAGE	^E
^H	LEFT	-SRCH	^T
^I	+TAB	-TAB	^X-^U
^J	DOWN	-WORD	^B
^K	UP	ALT	^X-^A or ^_ or ^/
^L	RIGHT	BSP	^C

Key (in alphabetical order) =	Function	Function (in alphabetical order)	Key
^M	RETURN	CHG WIN	^X-^W
^N	+WORD	CLOSE	^V
^O	OPEN	CMD	^A or NULL or ^@
^P	PICK	CTRLCHAR	^X-^C
^Q	(not used)	DELCHAR	^W
^R	+PAGE	DOWN	^J
^S	(not used)	ERASE	^X-^E or ^^

^T	-SRCH	HOME	^G
^U	MARK	INSERT	^Z or ESC or ^[
^V	CLOSE	INT	^\
^W	DELCHAR	JOIN	^X-^J
^X	(extend)	LEFT	^H
^Y	+SRCH	MARK	^U
^Z	INSMODE	OPEN	^O
^[	INSERT	PICK	^P
^\	INT	REPL	^X-^R or ^]
^]	REPL	RETURN	^M
^^	ERASE	RIGHT	^L
^_	ALT	SPLIT	^X-^B
^/	ALT	TABS	^X-^T
^X-^A	ALT	UP	^K
^X-^C	CTRLCHAR	WIN LEFT	^X-^H
^X-^E	ERASE	WIN RIGHT	^X-^L
^X-^H	WIN LEFT		
^X-^J	JOIN		
^X-^L	WIN RIGHT		
^X-^R	REPL		
^X-^B	SPLIT		
^X-^T	TABS		
^X-^U	-TAB		
^X-^W	CHG WIN		

ENVIRONMENT VARIABLES

`TERM` The `TERM` environment variable defines the terminal type. If there is no compiled in code for the terminal `e` will look in `$TERMINFO`/*first_letter_of_$TERM*/`$TERM` for a definition of the terminal.

`EKDB` The environment variable `EKBD` defines the keyboard type if different from the `TERM` type.

`editalt`
The `editalt` environment variable will be used as the name of the alternate file. If an alternate filename is given on both the command line and with the `editalt` variable, the `editalt` name wins (this is the reverse of what it should be and will change in future versions).

`EKBFILE`
The `EKBFILE` environment variable will be used as the name of a special keyboard file used by the Rand editor. The default meaning of keys are listed below.

`EEDFILE`
The Rand editor sets the environment variable `EEDFILE` to the filename that is just edited in the current session.

FILES

`e` creates and maintains several ".e" files. They are used to keep track of the current editing session and can normally be ignored by the user. The following information is for the curious.

`.es1` File `.es1` contains state information on the last edit session. It is created or updated at the end of a normally terminated edit session.

`.ec1` File `.ec1` indicates the last edit session in this directory crashed.

`.ek1` File `.ek1` contains a history of the last edit sessions keyboard inputs. The `.ek1` file is updated at regular intervals so that a minimum of work is lost on edit or system crashes.

`.ek1b`
File `.ek1b` is the `.ek1` file prior to a recovery replay.

`.ek1*`
The presence of any `.ek1*` file indicates that the last session in this directory abnormally terminated and that the next edit session will do recovery processing unless option `–norecover` is used. File `.ek1` means abort, `.ec1` means crash.

Note that if the login user is not the owner of the current directory the above files will be suffixed with the login users name. If the user has no write permission in the current directory, these files are created in the directory `/usr/etmp`.

There are also a few utility files in the directory `/etc/e`.

`#p` The pick file `#p` is re-created for each editor. It is used to store all text portions you can plach in the pick buffer during the editor session. The file is continually updated on the disk, and it can be used for recovery in the event of an abnormal termination.

During the editor session you can display this file as an alternate text in your window or another window to copy once more any text fragments you once had in the pick buffer.

`#o` The open file `#o` is also re-created for each editor session. It is used to store all text fragments you place in the close or erase buffer during the editor session or that were processed with the `fill`, `justify` or `center` command (in the original condition). The file is continually updated on the disk and can be used for recovery in the event of abnormal termination.

During the editor session, you can display this file as an alternate text in your window or another window to copy in their original condition any text fragments you have processed in this way or deleted.

NOTES

By far the worst problem with `e` is its treatment of tabs. `e` is a "white space" editor, all white space is considered equal. On any line that `e` modifies, it will compress multiple initial blanks to tabs, strip trailing blanks, and convert any embedded tabs to blanks. This is acceptable for most editing, but some programs expect real tab characters as input, for example `tbl`(1M) and some `nroff`(1M) commands.

echo(1)

NAME

`echo` - echo command line arguments

SYNOPSIS

`echo` [*argument* ...]

DESCRIPTION

The utility `echo` writes its arguments to standard output and terminates. The following takes place before the output is generated:

1. As with every other command, the shell interprets the command line arguments and then passes the edited arguments to `echo` [see `sh(1)`]. The argument separators recognized by the shell are blanks and tabs.
2. `echo` interprets any remaining special characters that control output (see description of *argument*).
3. `echo` displays processed arguments as follows:
   - Each argument is separated from the next by a blank even if you have entered more than one argument separator between individual arguments in the call.
   - A newline character is output at the end of the last argument.

You can use `echo` to

- examine values of shell variables,
- generate messages in shell scripts and thus test how they function,
- test how the shell interprets a command call without the command being executed,
- redirect data to a pipe and test how the pipe processes this input.

Besides the shell built-in `echo`, there is also a command called `/usr/bin/echo`. The shell generates a new process to execute `/usr/bin/echo`. Otherwise, `/usr/bin/echo` operates in the same way as `echo`.

OPERANDS

argument

Any string delimited by blanks or tabs. The last argument is terminated by means of a command separator.

argument not specified: `echo` produces only a blank line.

You may specify any number of arguments, provided they are separated by at least one tab or blank character.

As with any other command, this string is also initially interpreted by the shell:

- If the string contains an asterisk, question mark, or [...], the shell replaces this string by all suitable file names in the current directory. If no suitable file name is found, the shell passes the string on to `echo` unaltered.
- '*string*'

  All shell metacharacters are escaped by the single quotes. The shell passes the string as an argument to `echo` without the quotes. All blanks, tabs and newline characters entered within single quotes are retained.
- `` `string` ``

  The shell executes *string* as a Reliant UNIX command and passes the output of this command to `echo`. Characters assigned to the environment variable `IFS` are interpreted by the shell in the output as argument delimiters. The default characters assigned to `IFS` are the blank, tab or newline characters.

- "*string*"

  The double quotes escape all shell metacharacters except for the backslash \, backquotes ` ...` and the dollar sign $. The shell passes the processed string to `echo` as an argument. All blanks, tabs, and newline characters within the double quotes are retained.

The shell built-in `echo` also interprets the control characters described below. Since the backslash has a special meaning for the shell, it must be escaped:

- by preceding it with another backslash \

  This also applies if the argument containing this control character is enclosed within double quotes.

- by single quotes '...'

  If the argument containing this control character is enclosed within single quotes, the backslash need not be escaped again.

Example:

Input	Output
`hello\\bi`	`helli`
`hello'\'bi`	`helli`
`'hello\bi'`	`helli`
`"hello\\bi"`	`helli`

The following control characters influence the output of `echo`, provided the backslash has been appropriately escaped:

`\a` (a – alert) warning, bell; a beep is usually heard when output appears on the screen, the character has no effect for printer output.

`\a` corresponds to CTRL-G

`\a` is only interpreted by the Korn shell `ksh`.

`\b` (backspace) `echo` sets the cursor position one column to the left. Any character that is entered after the control character will overwrite the character at which the cursor is located.

`\b` corresponds to BACKSPACE or CTRL-H

`\c` `echo` prints the arguments entered up to this control character, omitting the newline and ignoring the remaining arguments.

`\f` (form feed) This control character is not converted if the output is written to the terminal.

If the output from `echo` is sent to a printer, for example, all arguments entered after this control character will be written on the next page.

`\f` corresponds to CTRL-L

`\n` (newline) All arguments that follow the newline character are written in the next line.

`\n` corresponds to RETURN or CTRL-J

`\r` (carriage return) `echo` sets the cursor position to column 1 of the current line. Characters entered after this control character overwrite any characters previously printed by `echo` in the columns of the current line.

`\r` corresponds to CTRL-M

`\t` Control character for tab; `echo` moves the cursor to the next tab stop. Characters entered after `\t` are written by `echo` from this column onward.

The setting of tab stops depends on the data terminal that is used. On a Siemens Nixdorf 97801 terminal, tab stops are set at the following columns:

1 9 17 25 33 41 49 57 65 73 79

`\t` corresponds to TAB or CTRL-I

`\v` Control character for vertical tab; `echo` sets the cursor to the next vertical tab stop. Vertical tabs are not usually set (depending on the type of terminal used).

`\v` corresponds to CTRL-K

`\\` Write a backslash character (\).

`\0`*n* *n* must be a one to three-digit octal number. `echo` outputs the corresponding character [see `ascii`(5) and `meta-ascii`(5)]. You can use this option to generate characters with a machine code higher than 7f (hexadecimal), for example, even if you do not have an 8-bit terminal.

Non-printing characters are represented by a smudge character (depending on the type of terminal used).

Caution:

If a character in octal notation is to be followed by an additional digit that is not part of the octal number, you must use the full three-digit form of the octal number.

Example:

Command	Result
`echo '\0337'| od -xc`	`0adf` (hex)
	`337 \n` (ascii)
`echo '\00337'| od -xc`	`371b 000a` (hex)
	`033 7 \n` (ascii)

If control characters of the above type are to be interpreted by neither the shell nor `echo`, they must be entered as follows:

Input	Output
`\\`	the backslash `\`
`\\\\b`	the string `\b`
`'\\b'`	the string `\b`
`"\\\\b"`	the string `\b`

The above example applies to the remaining strings as well.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

Example 1

The following interactive session will demonstrate when blanks, tabs and newline characters are retained:

```
$ echo Today is         Monday.□
Today is Monday.
```

In this case `echo` receives three arguments and returns them with one blank between each.

```
$ echo "Today is          Monday."□
Today is         Monday.
```

All blanks entered within the double quotes are retained.

```
$ echo `ls p*`□
```

```
pers pp ppp probe punch pwd pwddr
$ echo "`ls p*`"
pers
pp
ppp
probe
punch
pwd
pwddr
```

Without double quotes, the shell removes all argument separators (except for one blank) from the output of the `ls` command and passes the output on to `echo`. The double quotes cause all blanks, tabs, and newline characters to be retained.

Example 2

User `anna` wants to know what value is assigned to the `HOME` variable:

```
$ echo $HOME
/home/anna
```

Example 3

The following line in a shell script will cause an error message to be written on standard error:

```
echo File $1 not found >&2
```

Example 4

The following line in a shell script will produce an acoustic signal before the error message:

```
echo \\07File $1 not found >&2
```

Example 5

A string constant is to be placed before the output of the `date` command:

```
$ echo "Today's date and time:  \\c"; date
Today's date and time:  Wed Mar  8 17:22:05 EST 1991
```

The control character `\c` prevents `echo` from interpreting the newline. The backslash must be escaped despite the double quotes.

NOTES

`echo` exists both as an external command (`/usr/bin/echo`) and as a built-in shell command, both in the Korn shell `ksh(1)` as well as in the Bourne shell `sh(1)`. The shell generates a new process to execute `/usr/bin/echo`.

Some differences in behavior may occur when using `echo`, depending on which shell is being used. The possible differences are not described specifically.

The external `/usr/bin/echo` command recognizes the `–n` option which suppresses a line feed in output [similar to the BSD `/usr/ucb/echo` command, see `echo(1-ucb)`].

SEE ALSO

`csh(1)`, `ksh(1)`, `sh(1)`, `ascii(5)`, `meta-ascii(5)`.

echo(1-ucb)
BSD Compatibility Package

NAME

`echo -` echo arguments

SYNOPSIS

`/usr/ucb/echo` [`–n`] [*arg* ...]

DESCRIPTION

`echo` writes its arguments separated by blanks and terminated by a new-line to standard output.

OPTIONS

`–n` Do not add the newline to the output.

EXAMPLES

The call

```
$ /usr/ucb/echo –n text\\a
```

prints the following to standard output:

```
text\a$
```

User's Reference Manual

ed(1)

NAME

`ed` - interactive line editor

SYNOPSIS

`ed` [`-s`] [`-p` *string*] [`-x`] [`-C`] [`--`] [*file*]

DESCRIPTION

`ed` is an interactive line editor. Furthermore, with the help of `ed` scripts (see **WORKING WITH ED SCRIPTS**), you can easily process several files with the same sequence of commands. `ed` can handle the output of the `diff -e` command [see `diff(1)`].

OPTIONS

`-s` The `-s` option suppresses the following default outputs:

- number of bytes processed by the `ed` commands:
  `e` (edit)
  `r` (read)
  `w` (write)
- the question mark character, which warns against inadvertent deletion of the buffer contents during execution of the `ed` commands:
  `e` (edit)
  `q` (quit)
- the exclamation mark ! used as an `ed` prompt after a `!` command.

The `-s` option corresponds to the old - option, which is still supported.

`-p` *string*
In *string* you can define the prompt that `ed` displays in command mode. *string* can be one or more characters.

`-p` *string* not specified:

`ed` does not display a prompt string.

`-x` Encryption option: `ed` executes its own `X` command and prompts you for a key. This key is then used to encrypt and decrypt text using the algorithm of the `crypt` command. `ed` checks whether or not the text to be read is encrypted. The temporary buffer file is likewise encrypted, using a transformed version of the key you enter here [see `crypt(1)`].

`-C` Encryption option; the same as the `-x` option, except that `ed` executes its `C` command. The `C` command is like the `X` command, except that all text read in is assumed to have been encrypted [see `crypt(1)`].

`--` If *file* begins with a dash (-), the end of the command-line options must be marked with `--`.

file The name of the file that you wish to process. `ed` copies the file into its internal buffer and saves *file* as the current file name.

file not specified:

You start by working on an empty buffer and only decide upon a file name when using the `w` *file* command to write the buffer contents into a file.

ED BUFFER

When `ed` is invoked, a buffer is opened.

If you have not entered a file name, the buffer will be empty. You can then fill it with text during your editor

session.

If you have named a file, a copy of the file is read into the buffer. During your editor session, you essentially process the contents of the buffer.

Before you exit the editor again, you must decide whether you want to save the newly created or modified buffer contents by writing them to a file.

If you wish to save the buffer contents, you use the `w` [*file*] (write) command to write the contents of the buffer back to the specified file (by default the one named when `ed` was invoked) and then exit the editor with the `q` (quit) or `Q` (Quit) commands or with the <CTRL-D> key.

If you do not wish to save the buffer contents, you can exit the editor without writing back the contents of the buffer with `w`. You do this by pressing `Q` or `q` twice. When you press `q` the first time, `ed` will issue a ? as a warning to prevent you from inadvertently deleting the buffer contents. The buffer will not be deleted unless and until you press `q` a second time. If you prefer, you can also enter `Q` or <CTRL-D> instead of the second `q`.

OPERATING MODES

`ed` provides you with two operating modes: command mode and input mode. `ed` enters command mode when it is called with `ed` [*file*] <RETURN>. In command mode you specify a command in a line and confirm your input by pressing <RETURN>.

Input mode is activated by means of one of the following commands:

`a` (append)

`i` (insert)

`c` (change)

(see **ED COMMANDS** below).

In input mode, all the input characters which follow, including various non-printing characters (e.g. key codes of cursor keys), are written to the working copy in the buffer. `ed` does not accept any commands in input mode. If you wish, you can define a prompt for command mode (see option `–p` and `ed` command `P`) so that you can instantly detect the mode in which you are currently working. You leave input mode either by pressing <DEL> or by entering a period (.) in the first column and hitting <RETURN>. When you press <DEL>, `ed` normally ignores all input since the last <RETURN> and displays a ? as a warning.

COMMAND STRUCTURE

For `ed`, there is a **current line** at all times. As a rule, the line last processed by a command represents the current line. If you do not specify another address in front of the commands, they will always refer to the current line.

Most `ed` commands have the following structure:

[*range*]*ed-command*[*parameter* ...] <RETURN>

range The *range* you enter identifies the lines in the buffer to which the `ed` command is to be applied. One or two addresses can be specified in *range*:

range = *address*
The line identified by *address* is selected.

range = *address1*,address2
limits (inclusive). The search for both addresses begins at the current line, which is not changed until commands are executed.

address2 must refer to a line that follows the line referenced by *address1* in the buffer. Otherwise, `ed` reports an error.

range = *address1*;*address2*
address1;*address2* identifies the range between the specified limits (inclusive). The search for *address1* begins at the current line. The new current line is set to the line identified by *address1*, and only then is *address2* calculated. You can use this feature to define the starting line for forward and backward searches (see **ADDRESSES**).

address2 must refer to a line that follows the line referenced by *address1* in the buffer. Otherwise, `ed` reports an error.

range not specified:

`ed` assumes the default address for each command; this address is described for each of the `ed` commands.

If `ed` requires no address but you have nevertheless specified one, `ed` reports an error.

If you have specified more addresses than necessary, `ed` uses the last address(es) specified.

ADDRESSES

Addresses are constructed as follows:

Address	Meaning
`.`	Current line
`$`	Last line
n	*n*th line
`'`*x*	The line marked with the letter *x*. *x* must be a letter in lowercase (see `k` command).
`/`*RE*`/`	A simple regular expression *RE* enclosed in /.../ [see `expressions(5)`] addresses the first line containing a character string that matches the regular expression, searching forward from the current line. If `ed` does not find a match in any line, the search wraps around and continues from the beginning of the file until a match is found or the current line is reached again. If the regular expression *RE* contains the delimiter / or ?, this must be escaped using \. The characters \n in a regular expression do *not* match newline characters in the searched text! A regular expression may only appear once in *range*. Thus, `/`*RE1*`/,/`*RE2*`/`, for example, will only address the first line that matches `/`*RE2*`/`.
`//`	A null regular expression // addresses the line matching the regular expression last specified.
`?`*RE*`?`	Like `/`*RE*`/`, except that the search begins at the current line and proceeds toward the beginning of the file. If the regular expression *RE* contains the delimiter / or ?, this must be escaped using \.
addr[+]*n*	*n*th line after the line identified by *addr*.
addr–*n*	*n*th line before the line identified by *addr*.
+*n*	*n* lines forward from the current line.
–*n*	*n* lines backward from the current line.
[*addr*]+ ... [*addr*]– ...	One line forward (+) or backward (–) from the line identified by *addr*. Each occurrence of + or – respectively increases or decreases the address specification by 1. Thus, ++ addresses the second line after the current line (2 lines forward).
`,`	A comma stands for the address pair 1,$ if followed by a command; if not, the last line is output.
`;`	A semicolon stands for the address pair .,$ if followed by a command; if not, the last line is output.

ED COMMANDS

The following list includes a systematic overview of all `ed` commands that you can enter in the command mode. The detailed command description that follows is arranged in alphabetical order.

Overview of the ed commands

Activate input mode

a	append	append text after addressed line
c	change	delete and replace
i	insert	insert text before addressed line

Output prompt in command mode		
P	prompt	use * as prompt

Undo commands		
u	undo	undo most recent command

Abort commands		
<DEL>	---	abort execution of command

Explain errors		
h	help	explain last error message
H	Help	Toggle help mode on and off. If help mode is on, error messages are printed for all subsequent ? diagnostics (see **ERROR MESSAGES**).

Modify text		
a	append	append text after addressed line
c	change	delete and replace
d	delete	delete lines from buffer
i	insert	insert text before addressed line

Output lines		
p	print	print addressed lines
l	list	output with non-printing characters in alternative representation or as octal numbers
address	---	output addressed lines
n	number	print indicated lines with line numbering
<RETURN>	---	output line following current line

Output line numbers		
address=	---	output addressed line number
n	number	output lines with line numbering

Move specified line ranges		
t	transfer	append a copy of addressed lines
m	move	move lines to after addressed line

Search and replace

s	substitute	search and replace

Join lines		
j	join	join contiguous lines

Mark lines		
k	mark	mark addressed lines

Process selected lines with commands		
g	global	apply command list globally to all lines that match the given */RE/*
G	Global	apply interactive command list globally to all lines that match the given */RE/*
v	vice-versa	like g, but for all lines that do not match */RE/*
V	Vice-versa	like G, but for all lines that do not match */RE/*

Change current file name		
f	file-name	change/display current file name

Execute shell commands		
!	———	send command to shell for interpretation

Read files into buffer		
e	edit	delete buffer and read named file into it
E	Edit	clear buffer without warning and reload original
r	read	read file into buffer

Encrypt and decrypt file contents		
C	crypt	specify key for reading
X	crypt	specify key for reading and saving

Save buffer contents		
w	write	write buffer contents into file
W	write	append buffer contents to file

Quit the editor		
q	quit	quit ed
Q	Quit	quit ed without warning
<END>	———	quit ed

DESCRIPTION OF THE ED COMMANDS

The square brackets [] are not to be entered. They merely indicate that the entry enclosed within them is optional.

As a rule, only one command may be entered per line. However, you can append the suffixes `l`, `n`, or `p` to the commands (with the exception of `e`, `f`, `r`, and `w`) if the functions described under `l`, `n`, and `p` are to be executed.

[*address*]`a`
text
.

(append) reads the *text* input and appends it to the line addressed in *address*. The current line is now either the last line of the inserted text or, if you have not entered any text, the addressed line. Address 0 is legal for this command: it causes the "appended" text to be inserted at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 2048 per line (including the newline character).

address not specified:

address = .

[*range*]`c`
text
.

(change) deletes the specified *range* and replaces these lines with the *text* input. The current line is now either the last line of the entered text or, if you have not entered any, the line following the deleted lines.

range not specified:

range = .,.

`C` Encryption command:

the `C` command is like the `X` command, except that all text read in with the `e` and `r` commands is assumed to have been encrypted (see option `-C`).

[*range*]`d`

(delete) deletes the specified *range*. The line after the last line deleted becomes the current line. If the deleted lines were at the end of the buffer, the new last line becomes the current line.

range not specified:

range = .,.

`e` [*file*]

(edit) deletes the entire buffer and reads in a copy of the contents of the named *file*. If the contents of the buffer have been modified but not saved with `w`, `ed` prevents inadvertent deletion of the buffer by first issuing a ? as a warning. If you now enter `e` again, the old buffer contents are deleted without further comment. The number of bytes read is output provided you did not call `ed` with the `-s` option. The current line is the last line of the buffer. The specified file name *file* is remembered for possible use as a current file name in subsequent `e`, `r`, and `w` commands. If *file* is replaced by an exclamation point !, the rest of the line is interpreted as a shell command and executed. The output of the shell command is read into the buffer. A shell command that is preceded by an ! is not stored as the file name.

file not specified:

file = current file name

`E` [*file*]

(Edit) behaves like `edit`, except that it overwrites the buffer without issuing a ? as a warning even if the buffer contents have been modified but not saved.

file not specified:

file = current file name

`f` [*file*]

(file) sets the current file name to *file*. The current file name is used by the commands `e`, `E`, `r` and `w`.

file not specified:

`ed` outputs the current file name.

[*range*] `g`/*RE*/*commandlist*

(global) first marks all lines containing a character string which matches the regular expression *RE*. *RE* is a simple regular expression [see `expressions(5)`]. Then *commandlist* is executed for each line marked, with the current line being set to the next marked line in each case.

A single command or the first in a *commandlist* must be entered in the same line as the `g` command. All lines in *commandlist* except the last one must end with \<RETURN>; the last command itself with <RETURN>.

The commands `a`, `i`, and `c` with their associated input *text* are allowed. All lines in *text* must also be terminated with \<RETURN>. The period "." usually used to terminate input can be omitted from the last line of *commandlist*.

An empty *commandlist* is equivalent to the `p` command.

The `g`, `G`, `v`, and `V` commands are not permitted in the *commandlist*.

The global command must not be combined with the `!` command.

range not specified:

range = 1,$

[*range*]`G`/*RE*/

(Global) is the interactive variant of the `g` command. First, every line that matches *RE* is marked. *RE* is a simple regular expression [see `expressions(5)`]. The first of the marked lines is printed. At the same time, this line becomes the current line.

You now have the option of specifying a command to be executed (other than `a`, `c`, `i`, `g`, `G`, `v`, or `V`). After the execution of that command, the next marked line is printed, and so on.

A newline acts as a null command; an ampersand & 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 pressing the <DEL> key.

range not specified:

range = 1,$

`h` (help) issues a short error message that explains the reason for the most recent ? symbol displayed on the screen. See **ERROR MESSAGES** for a list of possible error messages.

`H` (Help) causes `ed` to enter a mode in which error messages are printed instead of the ? symbol for all errors that follow. It will also explain the previous ?, if any. You can deactivate the help feature by calling the `H` command again.

The `H` mode is normally off. See **ERROR MESSAGES** for a list of possible error messages.

[*address*]`i`
text
.

(insert) inserts the given *text* before the line referenced by *address*. The last inserted line of text, if any, becomes the current line; otherwise, the addressed line does. This command differs from the `append` command only in the placement of the entered *text*. Address 0 is not legal. The maximum number of characters that may be entered from a terminal is 2048 per line (including the newline character).

address not specified:

address = .

[*range*]`j`

(join) joins all lines in the specified *range* into one line by removing the appropriate newline characters. If you have only specified one address, nothing happens. The new line becomes the current line.

range not specified:

range = .,.

[*address*]k*x*

(mark) marks the line referenced by *address* with the letter specified in *x*, where *x* must be in lowercase. The marker itself is not output. The marked line can then be addressed by using `'`*x* (single quote *x*). The current line is unaffected.

address not specified:

address = .

[*range*]l

(list), in contrast to p, outputs the specified *range* as follows: some non-printing characters are output in alternative representation (e.g. tab characters), the remaining non-printing characters are output as octal numbers. Overlength lines are folded into several lines, each terminated by the line continuation character \. Each line must end with $. An l command can be appended to any command other than e, f, r, or w.

The following alternative representations are used:

\\ Backslash (for distinguishing octal characters)

\a Warning, bell

\b Backspace

\f Form Feed

\r Carriage Return

\t Tab

\v Vertical tab

range not specified:

range = .,.

[*range*]m*address*

(move) re-positions the lines in the specified *range* after the line addressed by *address*. The last of the shifted lines becomes the current line. If you specify a value of 0 for *address*, the *range* is moved to the beginning of the file. If *address* falls within the *range* of moved lines, ed issues an error message.

range not specified:

range = .,.

[*range*]n

(number) prints the *range* of addressed lines, preceding each line by its line number and a tab character. The last line to be printed becomes the new line. The n command can be appended to any command other than e, f, r, or w.

range not specified:

range = .,.

[*range*]p

(print) prints the *range* of addressed lines. Non-printing characters are not output unchanged. Overlength lines are continued in the next line, i.e. are not identifiable as such. The current line is the last line printed. The p command can be appended to any command other than e, f, r and w. For example, dp deletes the current line and prints the new current line.

range not specified:

range = .,.

P (Prompt) causes ed to use an asterisk * or the defined prompt string as a prompt in command mode (see option –p). You can deactivate this mode again by calling P a second time. This mode is normally deactivated.

q (quit) terminates ed. If you have changed the buffer contents since the last time the buffer was saved or overwritten, but have not yet written these changes to a file with w, ed outputs a ? as a warning (to prevent inadvertent deletions) and waits for further input. By entering <CTRL-D>, Q, or the q command

a second time, you can now exit `ed` without further warnings and without saving the buffer.

Caution:

If you continue after the first `q` with actions that do not change the buffer contents, no new warning will be issued when you press `q` again. `ed` will be terminate without saving the buffer.

`Q` (Quit) terminates `ed` immediately without warning even if you have changed the buffer contents since last saving or overwriting and have not yet written these changes to a file with `w`.

[*address*]`r` [*file*]

(read) reads the named *file* and inserts its contents after the line identified by *address*.

The address 0 is legal for this command and causes the file contents to be written at the start of the buffer. If the read was successful, the number of bytes read is output unless you called `ed` with the `-s` option. The current line is the last line read in. The currently remembered file name is not changed to *file* unless you invoked `ed` without a file name and *file* is the very first file name mentioned since `ed` was invoked. If *file* is replaced by the ! character, the rest of the line is interpreted as a shell command and executed. The output of this command is then read. Such a command is not remembered as the current file name.

address not specified:

address = $

file not specified:

file = current file name

[*range*]`s`/*RE*/*replacement_string*/[`g`|*n*]

(substitute) searches each line in *range* for strings which match *RE*. *RE* is a simple regular expression [see `expressions(5)`]. On *each* line in which a match is found, strings that match *RE* are replaced by *replacement_string*: without `g` and *n*, only the first occurrence of the matched string is replaced; with `g`, all matches are replaced; and with *n*, the nth occurrence on the line is replaced. *n* is a positive integer between 1 and 512. If `ed` does not find a matching string, it returns a ? to indicate an error. To delimit the regular expression *RE* from the `s` command and the

replacement_string, any other character except the blank or newline can be used instead of /. The character selected is recognized as a delimiter by virtue of coming immediately after `s`. Afterwards, the current line is the line in which the last substitution took place.

Metacharacters in the replacement string

Metacharacters

Meaning

`&` is replaced by the string which matches regular expression *RE* in a successful search.

`\`*n* is replaced by the character string matching the *n*th regular subexpression, delimited by \(...\), of *RE* where *n* is a decimal digit. If nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(from left to right.

`%` is replaced by the replacement string from the most recent `s` command if *replacement_string* consists solely of the `%` character.

The special meaning of these characters can be suppressed by preceding each of them with a backslash \.

Splitting a line in the replacement string

To split a line you can include an escaped newline character, i.e. \<RETURN>, in *replacement_string*. This type of substitution command cannot be used in a *commandlist* with a `g` or `v` command.

range not specified:

range = .,.

[*range*]`t`*address*

copies the addressed *range* after the specified line *address*. 0 is allowed for *address*. The current line is the last of the copied lines.

range not specified:

range = .,.

`u` (undo) nullifies the effect of the most recent command that modified anything in the buffer. The following commands can be undone: `a`, `c`, `d`, `g`, `i`, `j`, `m`, `r`, `s`, `t`, `v`, `G`, and `V`.

[*range*]`v`/*RE*/*commandlist*

(vice versa) causes *commandlist* to be executed on all lines containing *no* string that matches the regular expression *RE*. *RE* is a simple regular expression [see `expressions(5)`]. `v` functions identically to the global command `g` with the selection criterion reversed.

`v` should not be combined with the `!` command.

range not specified:

range = 1,$

[*range*]`V`/*RE*/

(Vice versa) is the interactive variant of the `v` command. You can use `V` to process all lines that contain *no* string matching the regular expression *RE*. *RE* is a simple regular expression [see `expressions(5)`]. `V` functions identically to the global command `G` with the selection criteria reversed.

range not specified:

range = 1,$

[*range*]`w` [*file*]

(write) writes the specified *range* into the named *file*. The currently remembered file name is not changed if previously set. If the file name has not been set, *file* is used as the new current file name. The old contents of *file* are overwritten in the process. If the named *file* does not exist, it is created. The current line remains unchanged. After a successful write operation the number of characters written is displayed, provided you did not invoke `ed` with the `-s` option.

If *file* is replaced by !, the rest of the line is interpreted as a shell command and executed. The standard input for the shell command is the *range* of addressed lines. Such a command is not remembered as the current file name.

range not specified:

range = 1,$

file not specified:

file = current file name

[*range*]`W` [*file*]

(Write) appends the specified *range* to the end of the named *file*. This command is the same as the `w` command above, except that `w` overwrites existing files. If *file* does not exist, it is created.

`X` Encryption command: `ed` prompts you for a key, which it uses in subsequent `e`, `r`, and `w` commands to decrypt and encrypt text using the algorithm of the `crypt` command. A null key turns off encryption. `ed` verifies whether or not text it reads in (`e` and `r`) is encrypted. The temporary buffer file is likewise encrypted, using a transformed version of the key you type in here (see the `-x` option).

[*address*]=

The line number of the line identified by *address* is output; the current line is not changed by this command.

address not specified:

address = $

address

The line identified by *address* is output.

`!`*command*

The remainder of the line after the `!` is interpreted and executed as a shell command. If an unescaped `%` character appears within the text of *command*, it is replaced by the remembered file name. `!!` causes

the last *command* to be repeated. In both cases, the expanded command line is echoed. On completion of the *command*, `ed` is reactivated. The current line remains unaffected. This command must not be combined with `g` and `v`!

<RETURN>
Entering <RETURN> alone in the command mode causes the line after the current line to be printed. This is the same as specifying .+1p. You can use this feature to step through the buffer.

The input of a command in command mode or a text line in input mode must be terminated by pressing the <RETURN> key.

<DEL> You can use the <DEL> key to interrupt a currently executing `ed` command or to cancel the entry of a line. `ed` will then respond by displaying a "?".

<CTRL-D>
The <CTRL-D> key has the same effect as the `q` command.

If the closing delimiter of a regular expression or of a replacement string (e.g. a /) is the last character before a newline, you may omit that delimiter. In such a case, the addressed line is printed. The following pairs of commands are equivalent:

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

WORKING WITH ED SCRIPTS

Since `ed` reads commands and text that is to be inserted from standard input, you can redirect the input to a file and have `ed` read the file instead. Thus

```
$ ed - file < ed_scriptfile > output
```

edits the named *file* and processes it with the `ed` commands stored in the specified `ed_scriptfile`. The `-s` option suppresses default output of message texts to the screen.

The advantage of using `ed scripts` is that they allow you to reproduce specific command sequences at any time and use them as often as required. Furthermore, this feature enables you to perform these tasks in a background process while you continue working at the terminal without interruptions:

```
$ ed file < ed_scriptfile&
```

If your `ed` script contains errors, `ed` will exit on encountering the first error.

EXIT STATUS

`0` if successful

`>0` if `ed` is called incorrectly or if a script is aborted due to incorrect use of `ed` commands.

ERROR MESSAGES

If you make a mistake when entering `ed` commands:

`?`

This means that the command contains syntax errors.

`?`*file*

The indicated *file* is not available or cannot be read.

More detailed information can be obtained by using the `h` and `H` commands. The commonest error messages are listed below. They are all self-explanatory:

```
Line out of range
Warning: expecting 'w'
```

```
No space after command
Unknown command
Bad range
Cannot open input file
Illegal or missing delimiter
Illegal suffix
Illegal or missing filename
No match
```

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

In regular bracketed expressions, the LC_COLLATE environment variable governs the scope of character ranges, equivalence classes and collating elements, while the LC_CTYPE environment variable governs the scope of character classes. LC_CTYPE also governs which characters the ed command l treats as non-printing.

If LC_MESSAGES, LC_COLLATE or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Example of an ed script:

The first three lines of a file are to be replaced by one line with the text "Addresses", and all occurrences of the word "Street" are to be replaced by "Plaza".

Contents of the edscript:

```
1,3c
Addresses
.
1,$s/Street/Plaza/g
w
q
```

Processing of a file with commands from edscript:

```
$ ed file < edscript
```

When ed reads commands from a file instead of the keyboard, the editor is exited as soon as ed encounters the first incomprehensible command.

Example 2

Example of a *Here* script [see sh(1)]:

In a set of files that are to be passed as command-line arguments to the script xy, the first three lines are to be replaced by one line with the text "Addresses", and all occurrences of the word "Street" are to be replaced by "Plaza".

Contents of the script file xy:

```
for i in $*
do
ed $i << screnD
1,3c
Addresses
```

```
.□
1,\$s/Street/Plaza/g□
w□
screnD□
done
```

Processing of the files `text1`, `text2` and `text3` with script `xy`:

```
$ sh xy text1 text2 text3
```

The string `<< screnD` after the `ed` call causes the shell to transfer the text up to the string `screnD` as input to `ed`. The second string `screnD` must exist as a single word in a line, without leading blanks. The special meaning of the `$` symbol must be escaped with a \ in the address specification 1,$. This is because the shell would otherwise interpret the following `s` as the name of a shell variable.

Example 3

The following example demonstrates and explains a number of `ed` commands:

`$ ed` – Call ed

`P` – Output prompt `*` in command mode

`*a` – Append to current line, in this case beginning of file

```
line1  – Input text
line2
line3
.
```

– Terminate input mode

`*1,$p` – Output line 1 through to last line

```
line1
line2
line3
*p
```

– Output current line (= last line processed)

```
line3
*n
```

– Output current line with line number

```
3 line3
*1,$n
```

– Output line 1 through to last line with line numbers

```
1 line1
2 line2
3 line3
*2c
```

– Replace line 2 with following input including tabs

`line2` <TAB> `line2` <TAB> `line2`

```
another line2
.
```

– Terminate input mode

`*p` – Output current line

```
another line2
*1,$n
```

– Output line 1 through to last line with line numbers

```
1 line1
2 line2 line2 line2
3 another line2
4 line3
*4s/3/4
```

– In line 4, search for the character 3 and replace it with 4

```
line4
```

```
*n
```

– Output current line with line number

```
4 line4
```

```
*1,$s/l/L/g
```

– Search from line 1 through to last line for all l characters and replace them with L

```
*1,$p
```

– Output line 1 through to last line

```
Line1
Line2 Line2 Line2
another Line2
Line4
```

```
*2l
```

– Output line 2 with non-printing characters as mnemonic overstrikes

```
Line2\tLine2\tLine2$
```

```
*
```

```
*2r file
```

– Read the contents of `file` into the buffer after line 2 and output the number of characters read in. `file` must already exist!

```
46
```

```
*1,$n
```

– Output line 1 through to last line with line numbers

```
1 Line1
2 Line2 Line2 Line2
```

– This line and the two which follow are out of `file`

```
3 line1 of file
4 line2 of file
5 last line of file
6 another Line2
7 Line4
```

```
*6,7c
```

– Replace lines 6 to 7 by the following input

```
very last line
```

```
.
```

– Terminate input mode

```
*1,$n
```

– Output line 1 through to last line with line numbers

```
1 Line1
2 Line2 Line2 Line2
3 line1 of file
4 line2 of file
5 last line of file
6 very last line
```

```
*q
```

– Attempt to quit editor with `q`

```
?
```

– Warning

```
*h
```

– Explain question mark

```
warning:expecting 'w'
```

– Explanation: `w` command expected by `ed` (to save contents of the buffer) was not entered

```
*w exp
```

– Write the contents of the buffer to the file `exp`

```
85
```

– Number of characters written is output

`*q` – Quit editor

`$` – Shell prompt

FILES

`ed.hup`
Data is saved in this file if `ed` receives the `SIGHUP` signal [see `kill(2)`].

`/var/tmp`
If this directory exists, it is used as the directory for storing the temporary work file.

`$TMPDIR`
If this variable is set and is not null, it is used instead of `/var/tmp` as the directory for storing the temporary work file.

`/tmp` `/tmp` is the directory used to store the temporary work file if the `TMPDIR` variable is not assigned the name of an existing directory and `/var/tmp` does not exist.

SEE ALSO

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

User's Reference Manual

edit(1)

NAME

`edit` - restricted version of ex text editor

SYNOPSIS

`edit` [`–r`] [`–x`] [`–C`] *file*

DESCRIPTION

`edit` is a variant of the text editor `ex(1)` and is quite easy to use. If you are a beginner or a casual user who only needs an editor occasionally, `edit` is probably the tool for you. The following brief introduction should help you get started with `edit`.

`edit` is a line and command editor. If you prefer to use a full-screen editor, it would be better to work with `vi(1)` or `ced(1)`.

OPTIONS

`–r` (restore) *file* is reconstructed if the editor aborts or a system crash occurs.

`–x` Encryption option: the file is encrypted as it is being written and will require an encryption key to be read. `edit` checks whether or not the text to be read is encrypted [see `crypt(1)`].

`–C` Encryption option: the same as `–x`, except that `edit` assumes that all files are encrypted.

file Name of the file that you wish create or process with the editor.

WORKING WITH EDIT

Editing existing files

To edit a file that already exists, open it with the command line:

`$ edit` *file*

`edit` will then create a copy of the file you wish to edit, write this copy to a buffer, and display the number of lines and characters in the specified *file*.

Creating a new file

To create a new file, select a name for the new file and issue the call:

`$ edit` *new_filename*

This command line causes the message "New file" to be displayed and an empty buffer to be opened. You can write text into the buffer using the `append` command.

Entering edit commands

After invocation, `edit` will prompt you to enter commands by displaying a colon as the input prompt.

If you are editing an existing file, the buffer will already contain lines on which you can use `edit` commands. To execute a command on a specific line, enter

line command <RETURN>

If you wish to execute a command on a range of lines (defined as *line*,*line*), enter the range and command as

line,*line command* <RETURN>

If you are creating a new file, the buffer will be empty when you call `edit`. The only useful commands at this stage are `append`, `insert` and `quit`.

Current line

Most `edit` commands use a preset *current line* unless you explicitly specify a line or a range. Thus, if you enter:

`:print` <RETURN>
(the <RETURN> key must be pressed to terminate all commands)

the current line will be printed. The command:

`:delete`

deletes the current line; `edit` will now print the new current line. When you start editing, `edit` makes the last line in the file the current line. In general, the last line to be addressed by a command becomes the current line.

Adding text – append

You use the `append` or `a` command when you want to create a new file or add lines to an existing one. `edit` will then continue reading lines from your keyboard until you enter a line that consists simply of a period in the first column:

`:append`*text* `.`□
`:`

If the file is new, the entered lines will be placed at the beginning; if not, they are appended to the current line. The last line you enter before the period becomes the new current line.

Adding text – insert

The `insert` or `i` command is like `append` except for the fact that it places the lines you enter *before*, rather than after, the current line.

`:insert`*text*□
`.`□
`:`

Line numbering

`edit` assigns a line number to each line in the buffer, starting with line number 1. When you issue the command:

`:1`

`edit` prints this line on the screen. The first line now becomes the current line, since it is the last line to have been processed by a command. If you subsequently enter:

`:delete`

`edit` deletes the current line (i.e. the first line) and converts what was originally the second line into the first. `edit` then outputs the new current line (the new first line) to show you where you are.

Changing text – substitute

The `substitute` or `s` command can be used to replace parts of text in the current line. Thus, if you enter:

`:s/`*old*`/`*new*`/`

the first occurrence of string *old* in the current line will be replaced by the string *new*. If the *old* or *new* string contains a slash, you must escape this character by preceding it with a backslash \.

Obtaining information – file

The *file* or `f` command tells you how many lines there are in the `edit` buffer and indicates whether you have changed this buffer:

`:file`

If you have changed the buffer contents, the message "[Modified]" will be issued.

Saving the buffer contents – write

The text that you have added to (or modified in) the buffer can be saved by using the `write` or `w` command, which writes the modified buffer contents back to the file you specified when calling the editor with the `edit file` command. The original contents of *file* will be overwritten!

```
:w
```

Quitting the editor – quit

After saving your work with the `write` command, you can exit the editor with the `quit` or `q` command:

```
:quit
```

If you have not changed the file during the `edit` session, it is not necessary (but does no harm) to `write` the file back. If you have modified the buffer and try to quit the editor without saving its contents (with `write`), `edit` issues the warning:

"No write since last change"

and waits for a further command.

You can still enter the `write` command at this stage and save the buffer contents if you wish. If not, enter □

```
:q!
```

The buffer is then irretrievably discarded, and you will be returned to the shell.

You are now in a position to edit files satisfactorily. However, if you plan to use `edit` frequently, you ought to become familiar with a few more features. Here are some additional options:

Replacing text – change

The `change` or `c` command can be used to replace the current line by a sequence of lines you supply. As in `append`, you have to terminate your new text by entering a period in the first column of a line:

```
:changetext .□
:
```

If you wish, you can change more than one line at a time by specifying the first and last line of a range, e.g. line 3 to line 5 (inclusive):

```
:3,5changetext .□
:
```

The 3 old lines (i.e. lines 3, 4, and 5) are now replaced by 1 new line containing the word `text`.

Displaying text – print

In the same way, you can also have several lines printed by using the `print` command. For example, the call:

```
:1,23print
```

will print the first 23 lines of the file.

Reversing commands – undo

The `undo` or `u` command reverses the effect of the last change made to the buffer. Thus, if you have issued a `substitute` command and made a change you do not want any more, you can enter:

```
:undo
```

and the old contents of the line will be restored. You can also `undo` an `undo` command if you change your mind again.

`edit` will give you a warning message when commands you invoke affect more than one line in the `edit` buffer.

Commands such as `write` and `quit` cannot be reversed.

Viewing lines

The next line is displayed if you simply press:

`1` <RETURN>

If you want to look at a number of lines (half a screen) you enter:

`:`<CTRL>□<D> (pressed simultaneously)

To examine a section of text that includes the current position, enter:

`:z.`

The current line will now be the last line that is displayed. You can also return to your position before the `z.` command by entering:

`:''`

To select different sections, you can give the `z` command with other following characters. For example:

`:z-`

outputs a full screen (24 lines). The last line to be output becomes the current line.

`:z+`

displays the next screen of information.

If you want to see less than a screenful, e.g. only 13 lines, you enter:

`:z.13`

The old current line will now be the seventh line on the screen, i.e. the middle line. If you enter an even number, `edit` will display one line less, so that the current line again appears exactly in the center of the screen. This method of giving counts also works for other commands; thus, you can delete 5 lines starting with the current line by entering the command:

`:delete 5`

Searching for text patterns

To find specific text in the file, you can use line numbers if you happen to know them. However, this is somewhat unreliable, since the line numbers change when you insert and delete lines. You can search forward for strings in the file by entering commands in the form:

`:/`*textpattern*`/`

To search backwards, you specify commands in the form:

`:?`*textpattern*`?`

If the end of the file is reached before the text to be located is found, the search wraps around and continues up to the line where you started the search.

If *textpattern* itself contains the delimiter / or ?, you must escape this character by preceding it by a backslash \. Two specially useful variants of the search command should be noted in this context. These features will allow you to locate a *textpattern* that is only at the beginning or only at the end of a line:

`:/^`*textpattern*`/`
: searches for *textpattern* at the start of a line, and

`:/`*textpattern*`$/`
: searches for *textpattern* at the end of a line.

The second / or ? may be omitted in these commands.

Special lines

The current line is designated as a . (period); the last line as the $ (dollar) symbol.

These designations are most useful when specifying ranges. For example,

`:.,$print`

will print the rest of the lines in the file. The command:

`:$delete`

deletes the last line in the file, no matter which line happens to be the current line.

It is also possible to specify positions relative to these designations, e.g.:

`:$-5` refers to the line 5 lines before the last one;

`:.+20` is 20 lines after the current line.

Current position

To find out the current line number, you can enter:

```
:.=
```

or

```
=
```

Specifying a range

If you wish to change, copy, move or print a section of text in the file, you can specify a range of lines before the commands themselves. For example:

```
:10,20delete
```

will delete the range of lines from line 10 to 20 (inclusive).

Moving lines

A range of lines can be moved as follows:

You first define the range (by using the = command), e.g.

```
10,20
```

and then call:

```
:10,20delete a
```

for this range. Lines 10 to 20 will be deleted and placed in a buffer named `a`. `edit` has 26 such auxiliary buffers. They are named `a,b,...z`.

This text can later be retrieved from the `a` buffer. The command:

```
:put a
```

fetches the contents of buffer `a` and copies them back to the file immediately after the current line. Buffer contents are only retained for the duration of an editor session and are overwritten when new text is placed in the same buffer.

Changing files

If you want to change the file as well, you enter:

`:edit` *other_file*

after buffering the lines. This command will cause `edit` to switch to *other_file*, and you can continue as described above.

Copying

To copy within the same file, e.g. to copy the range 10,20 to the end of the file, you enter:

```
:10,20move $
```

In this case you do not need the auxiliary buffer.

If you wish to copy lines to another file, use the command `yank` instead of `delete`. The remaining steps are the same as those indicated for moving lines.

OVERVIEW OF EDIT COMMANDS

The following list contains all `edit` commands in alphabetical order:

`a`	`append`	append text after addressed line
`c`	`change`	delete and replace lines
`d`	`delete`	delete line (range) or delete and write into a named buffer
`f`	`file`	print number of lines in the buffer
`i`	`insert`	insert text before addressed line
`m`	`move`	move specified range of lines

n1[,*n2*]	---	print line with number *n1* or range of lines from *n1* to *n2*
p	print	print current line
put	put	write content of a specified (named) buffer to the file copy
q	quit	quit edit
q!	quit	quit edit without warning
s	substitute	search and replace
u	undo	undo most recent command
w	write	copy buffer content into file
y	yank	write line (range) to (named) buffer
z	---	display surrounding lines
.=	---	display current line number
<RETURN>	---	jump to the next line in the buffer
<CTRL-D>	---	fill half the screen with lines
/*text*[/]	---	search for *text*
/^*text*[/]	---	search for *text* at start of line
/*text*$[/]	---	search for *text* at end of line
?*text*[?]	---	search backwards for *text*
?^*text*[?]	---	search backwards for *text* at start of line
?*text*$[?]	---	search backwards for *text* at end of line

EXIT STATUS

Always 0.

ERROR MESSAGES

`At end-of-file`

The end of the file has been reached.

`Badly formed address`

You have used an illegal format when specifying the address(es), e.g. 5.7 instead of 5,7.

`File is read only`

You only have read permission for this file [see chmod(1)] and are therefore not allowed to write the buffer contents to it.

`First address exceeds second`

You have specified an illegal range of lines; the second address is located before the first.

`Not an editor command`

The command you have used is not an `edit` command.

`Nothing in register b`

The buffer you have tried to use is empty.

`Not that many lines in the buffer`

The address you have specified is too high; the `edit` buffer does not contain that many lines.

`No write since last change (:quit! overrides)`

Warning: You have made changes to the buffer and have not yet saved these changes with `write`.

EXAMPLES

The following example demonstrates and explains a number of `edit` commands:

```
$ edit officearts
```
– Call `edit` with new file name

```
"officearts" [New file]
```
– Message from `edit`

`:i` – Insert before current line

```
No lines in the buffer
```
– Buffer is empty

`:a` – Insert after current line (= beginning of file)

```
Pencil
```
– Input text

```
Eraser
Binder
.
```
– Terminate input

`:p` – Output current line

```
Binder
```
– Current line (= last line processed)

`:1,$p` – Output first line through to last line

```
Pencil
Eraser
Binder
:3d
```
– Delete third line

```
Eraser
```
– New current line (= last line) is output

`:1,$p` – Output first line through to last line

```
Pencil
Eraser
:2i
```
– Insert before second line

```
Felt-tip pen
Pencil sharpener
.
```
– Terminate input

`:1,$p` – Output first line through to last line

```
Pencil
Felt-tip pen
Pencil sharpener
Eraser
:1,2delete a
```
– Delete lines 1 and 2 and write them to buffer `a`

```
2 lines deleted
```
– Confirmation

```
Pencil sharpener
```
– New current line (= new first line) is output

`:w` – Save the contents of the editor buffer by writing them back to the file specified when calling `edit`

```
"officearts" [New file] 2 lines, 24 characters
```

```
:1,$p
```
– Output first line through to last line
```
Pencil sharpener
Eraser
:edit writing
```
– Edit new file `writing`
```
"writing" No such file or directory
```
– Confirmation that this is a new file
```
:put a
```
– Write to buffer `a`
```
2 lines put
```
– Confirmation
```
Felt-tip pen
```
– Current (= last) line of `writing` is output
```
:1,$p
```
– Output first line through to last line
```
Pencil
Felt-tip pen
:w
```
– Write the contents of the editor buffer to the file last edited, in this case `writing`
```
"writing" [New file] 2 lines, 20 characters
:a
```
– Append to current (= last) line
```
Ball-point pen
```
– Input text
```
 .
```
– Terminate input
```
:q
```
– Attempt to quit editor
```
No write since last change (:quit! overrides)
```
– Warning that the contents of the editor buffer have not been saved with `w` since they were last modified
```
:w
```
– Write the contents of the editor buffer to the last file edited, in this case `writing`
```
"writing" 3 lines, 35 characters
```
– Confirmation
```
:q
```
– Quit editor
```
$
```
– Shell prompt

SEE ALSO

`ced`(1), `ed`(1), `ex`(1), `vi`(1).

User's Reference Manual

egrep(1)

NAME

`egrep` - search for extended patterns (extended grep)

SYNOPSIS

`egrep` [*option ...*] [*patternlist*] [*file ...*]

DESCRIPTION

`egrep` reads lines from one or more files or from standard input and compares these lines with the specified patterns. Unless told otherwise (by options), `egrep` copies every line that matches one of the patterns to standard output.

`egrep` permits the use of extended regular expressions in the specified pattern [see `expressions(5)`].

If you specify more than one input file, the relevant file name will be displayed before each output line.

The `egrep` patterns to be used when comparing input lines may be specified via *patternlist*, the `–e` *patternlist* option, or the `–f` *patternfile* option. At least one of these three arguments must be specified, multiple specifications are allowed. Multiple specifications of these arguments are ORed.

OPTIONS

The options `–c`, `–l`, and `–q` are mutually exclusive.

No option specified:
`egrep` outputs all lines that match at least one of the patterns specified in *patternlist*. If you specify more than one input file, each output line will be preceded by the name of the file in which the line was found.

`–b` (block) Each output line is preceded by the number of the block in which it was found.

Each file is made up of 512-byte blocks which are numbered consecutively from 0.

The `–b` option is sometimes useful in locating disk block numbers by context (see the \24offset argument for the `od(1)` command, for example).

`–c` (count) `egrep` outputs only the number of lines found (i.e. the lines that `egrep` would have displayed without the `–c` option, see *Example 4*); the lines themselves are not displayed.

`–e` *patternlist*
(expression) Synonymous with *patternlist*. The `–e` option must always be specified if the first expression in *patternlist* begins with a minus sign. When used in conjunction with the `–e` option, a *patternlist* of this type is not interpreted as an option itself but as a list of patterns which `egrep` is to use in searching for matching input lines. Multiple specification of `–e` options are allowed.

`–f` *patternfile*
(file) `egrep` reads the pattern list from the file named *patternfile*. Each line in *patternfile* is interpreted as an extended regular expression. Multiple specifications of `–f` options are allowed.

`–h` (hidden) When searching multiple files, `egrep` does not write the file name before each output line.

`–i`
`–y`
(ignore) `egrep` does not distinguish between uppercase and lowercase.

`–l` (list) `egrep` simply outputs the names of files that contain at least one of the matching lines. (These are the lines that `egrep` would output if the `–l` option were omitted, see *Example 5*.) Each file name is printed just once. The lines themselves are not displayed.

`–n` (number lines) Each output line is preceded by its line number in the relevant input file. Line numbering starts at 1. If `egrep` is reading from standard input, the line number refers to the standard input.

`–q` (quiet) `egrep` does not produce any output, but returns only the corresponding exit status.

`-v` (vice versa) `egrep` outputs all lines that *do not* match any of the specified patterns.

In conjunction with option `-c`: `egrep` prints only the number of lines that do not match.

In conjunction with option `-l`: `egrep` only outputs the names of files containing such lines.

In conjunction with option `-q`: `egrep` returns only the corresponding exit status.

`-x` (exact) `egrep` only outputs the lines containing one of the specified patterns and no other characters.

In conjunction with option `-c`: `egrep` prints only the number of such lines.

In conjunction with `-l`: `egrep` only outputs the names of files containing such lines.

In conjunction with option `-q`: `egrep` returns only the corresponding exit status.

`--` End of the list of options. Must be specified if one of the arguments begins with `-`.

patternlist

A list of extended regular expressions that `egrep` is to use in searching for matching input lines [see `expressions(5)`]. Individual regular expressions must be separated by the newline character. Any newline character within *patternlist* is interpreted like an OR separator (|) in an extended regular expression.

Regular expressions of the type `(r|s)` can also be specified without the parentheses: `r|s` (see *Example 1*).

If *patternlist* contains newline characters or other characters that have a special meaning for the shell, you must enclose the specified *patternlist* in single quotes: '*patternlist*'.

file Name of the file that `egrep` is to scan. You may name any number of files.

file not specified: `egrep` reads its input from standard input.

grep, fgrep and egrep

The `grep`, `fgrep` and `egrep` commands perform similar functions and are largely identical in terms of usage. The following section lists the most important differences between these commands.

`grep` processes simple regular expressions if the options `-E` or `-F` are not specified.

`fgrep` only processes character strings. `fgrep` can search for a large number of character strings very efficiently: `fgrep` searches each individual line for all of the character strings. `fgrep` behaves in a similar way to `grep -F`.

`egrep` processes extended regular expressions. These include simple regular expressions, with the following exception: The simple regular expressions \(*regexpression*\) and \{...\} have no special meaning for extended regular expressions and are therefore not processed by `egrep`. `egrep` behaves in a similar way to `grep -E`.

EXIT STATUS

`0` Matching lines found

`1` No matching lines found

`>1` Syntax error or "File cannot be opened". This exit status remains valid even if lines have been found in other input files.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_COLLATE` environment variable governs the collating sequence.

`LC_CTYPE` governs character classes and character conversion (shifting, option `-i`/`-y`).

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The files `customer1` and `customer2` will be used as a basis for all following examples. Their contents are given below:

`customer1`:

```
080685    999.98  20 Units  Item  038   Nicholson Ltd.
120387   1240.25   3 Units  Item  023   Robinson Ltd.
180588    330.87   1 Units  Item  332   Robinson Ltd.
```

`customer2`:

```
morrow lance, 86 sherwood street, london w1
robinson peter, 16 garden hill, london ec3
```

Example 1

Output lines that match a pattern (without an option and with option `-i`):

```
$ egrep Robinson customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Robinson Ltd.
customer1:180588    330.87   1 Units  Item  332   Robinson Ltd.
```

If you also wish to find lines containing the word `robinson` in lowercase you enter:

```
$ egrep -i robinson customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Robinson Ltd.
customer1:180588    330.87   1 Units  Item  332   Robinson Ltd.
customer2:robinson peter, 16 garden hill, london ec3
```

More complicated patterns can be set up with the help of regular expressions, e.g.: Display lines which contain the string `Nicholson` or `Robinson`:

```
$ egrep '(Nichol|Robin)son' customer1 customer2
customer1:080685    999.98  20 Units  Item  038   Nicholson Ltd.
customer1:120387   1240.25   3 Units  Item  023   Robinson Ltd.
customer1:180588    330.87   1 Units  Item  332   Robinson Ltd.
```

Instead of the regular expression `(Nichol|Robin)son` you could also use the following regular expression:

```
(Nicholson|Robinson)
```

In this case you can leave out the parentheses;

```
Nicholson|Robinson
```

The OR operator (|) in the last expression `Nicholson|Robinson` could also be replaced by a newline character (see *Example 2*).

Example 2

Using several patterns (without an option and with option `-f`):

```
$ egrep '^1
> 1$' customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Robinson Ltd.
customer1:180588    330.87   1 Units  Item  332   Robinson Ltd.
customer2:morrow lance, 86 sherwood street, london w1
```

Alternatively, you could write both patterns into a file called `names` (each pattern in a separate line) and then call `egrep` as follows:

```
$ egrep -f names customer1 customer2
```

The same result is obtained when the newline character that separates the patterns `^1` and `1$` is replaced by the OR operator:

```
$ egrep '^1|1$' customer1 customer2
```

Example 3

Output lines that match *none* of the specified patterns (option `-v`):

```
$ egrep -v '^1
> 1$' customer1 customer2
```

```
customer1:080685    999.98  20 Units  Item  038   Nicholson Ltd.
customer2:robinson peter, 16 garden hill, london ec3
```

The above call thus yields all lines that neither begin nor end with 1. The same result can also be obtained with the following call (see *Example 1*):

```
$ egrep -v '^1|1$' customer1 customer2
```

Example 4

Display the number of lines found (option `-c`):

To begin with, the number of lines that start with 1 are to be output for each input file.

```
$ egrep -c '^1' customer1 customer2
customer1:2
customer2:0
```

The number of lines that *do not* begin with 1 are now to be displayed.

```
$ egrep -c -v '^1' customer1 customer2
customer1:1
customer2:2
```

Example 5

Display file names only (option `-l`):

The names of files containing lines that begin with 1 are to be output first.

```
$ egrep -l '^1' customer1 customer2
customer1
```

The names of files containing lines that do not start with 1 are displayed next.

```
$ egrep -l -v '^1' customer1 customer2
customer1
customer2
```

Example 6

Display found lines with relevant line numbers (option `-n`):

```
$ egrep -n -i robinson customer1 customer2
customer1:2:120387   1240.25   3 Units  Item  023   Robinson Ltd.
customer1:3:180588    330.87   1 Units  Item  332   Robinson Ltd.
customer2:2:robinson peter, 16 garden hill, london ec3
```

SEE ALSO

ed(1), fgrep(1), grep(1), sed(1), sh(1), expressions(5).

User's Reference Manual

enable(1)
AT&T Spooler only

NAME

`enable, disable -` enable/disable LP printers

SYNOPSIS

`enable` *printers*

`disable` [*options*] *printers*

DESCRIPTION

The `enable` command activates the named *printers*, enabling them to print requests submitted by the `lp` command. If the printer is remote, the command will only enable the transfer of requests to the remote system; the `enable` command must be run again, on the remote system, to activate the printer. (Run `lpstat –p` to get the status of printers.)

The `disable` command deactivates the named *printers*, disabling them from printing requests submitted by `lp`. By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class of printers. If the printer is remote, this command will only stop the transmission of jobs to the remote system. The `disable` command must be run on the remote system to disable the printer. (Run `lpstat –p` to get the status of printers.)

OPTIONS

Options for use with `disable` are:

`–c` Cancel any requests that are currently printing on any of the designated printers. This option cannot be used with the `–W` option. If the printer is remote, the `–c` option will be silently ignored.

`–r` *reason*
Assign a *reason* for the disabling of the printers. This *reason* applies to all *printers* specified. This *reason* is reported by `lpstat –p`. *reason* must be enclosed in quotes if it contains blanks. The default reason is `unknown reason` for existing printers, and `new printer` for printers just added to the system but not yet enabled.

`–W` Wait until the request currently being printed is finished before disabling the specified printer.

This option cannot be used with the `–c` option. If the printer is remote, the `–W` option will be silently ignored.

FILES

`/var/spool/lp/*`

SEE ALSO

`lp(1), lpstat(1).`

User's Reference Manual

env(1)

NAME

`env` - set environment for command execution

SYNOPSIS

`env` [`–i`] [`––`] [*name=value ...*] [*command* [*arg ...*]]

DESCRIPTION

The `env` command can be used to display current environment variables and their values or to set them for the execution of a specific command. `env` inspects the current environment, modifies it according to the assignment in *name=value*, and then executes the command in the modified environment. The existing specifications for *name* and *value* are overwritten by the new ones, and the new arguments are merged into the inherited environment before the command is executed. The valid environment for the execution of *command* thus consists of the new specifications together with any unmodified environment variables.

If no command is specified, `env` prints the resulting environment.

OPTIONS

`–i` The original environment is ignored; *command* is then executed with exactly the environment specified by the arguments.

The `–i` option corresponds to the old – option, which will continue to be supported.

`––` End of the list of options if the first argument begins with `–`.

name=value

name specifies the name of a variable that is to be valid for *command*.

value is the value of *name* which is to apply to *command*.

command

Name of the command or shell script which you would like to have executed in the defined environment.

arg Argument, e.g. positional parameters or user-defined variables which can be passed to *command*.

EXIT STATUS

`0` `env` executed successfully

`1-125`

Error

`126` The specified *command* exists, but cannot be invoked.

`127` The specified *command* cannot be found.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Output the current values of environment variables:

```
$ env
```

```
DRU01=-ws=G01
HOME=/home/sysiphus
LANG=En_US.ASCII
LOGNAME=sysiphus
MAIL=/var/mail/sysiphus
PATH=::/usr/bin:/usr/sbin:/opt/bin:
PS1=$
SHELL=/sbin/sh
TERM=97801
TERMCAP=/etc/termcap
TTY=/dev/004
TZ=MET-1MDT
USER=sysiphus
```

Example 2

Output the modified values of environment variables:

```
$ env PATH=$HOME/proc
DRU01=-ws=G01
HOME=/home/sysiphus
LANG=En_US.ASCII
LOGNAME=sysiphus
MAIL=/var/mail/sysiphus
PATH=/home/sysiphus/proc
PS1=$
SHELL=/sbin/sh
TERM=97801
TERMCAP=/etc/termcap
TTY=/dev/004
TZ=MET-1MDT
USER=sysiphus
```

The `PATH` environment variable has been modified.

Example 3

Output modified values of environment variables using the – option:

```
$ env - PATH=$HOME/proc
PATH=/home/sysiphus/proc
```

The original environment is ignored.

Example 4

Invocation of the file `ardor`, located in `some/sisyphus/SAYINGS`, i.e. in a subdirectory of the `HOME` directory.

Contents of file `ardor`:

```
echo "$1 $2 $3 $4 $2 $3 $1 love $4!"
```

Now call `ardor` from any location in your file tree with the arguments `I know that you`:

```
$ env PATH=$HOME/SAYINGS ardor I know that you
I know that you know that I love you!
```

With the new definition of the `PATH` variable, you effectively define the location where the entered command is to be sought (the file `ardor` in our case), i.e. in a subdirectory of the `HOME` directory, which you specify with the value of the variable `HOME` (`$HOME`).

The character strings `I`, `know`, `that`, and `you` are passed as arguments to the positional parameters `$1`, `$2`, `$3` and `$4`.

The contents of `ardor` are correctly executed here because `echo` is a built-in command of the `sh` shell. Due to the change in the `PATH` variable, all Reliant UNIX commands in `/usr/bin`, `/usr/sbin` or `/opt/bin` can no longer be found. To enable the execution of Reliant UNIX commands again, the `PATH` variable must be

modified as shown in the example below:

Example 5

Call the file `delcopy`, which is located in the directory `/home/sisyphus/proc`. This file contains a script that compares two files and deletes one of them if both files are identical.

Contents of the file `delcopy`:

```
if cmp -s $1 $2
then
rm $2
fi
```

Call the `delcopy` file from any location in your file tree with the arguments `file1` and `file2`:

```
$ env PATH=$PATH:$HOME/proc delcopy file1 file2
```

In this case, the new path is appended to the original one. This ensures that the `delcopy` script can be executed along with all Reliant UNIX commands contained in it. If only the path for `delcopy` were specified, the following error message would be issued.

```
/home/sisyphus/proc/delcopy: cmp: not found
```

SEE ALSO

sh(1), set(1), exec(1), profile(4), environ(5).

eval(1)

NAME

`eval` - evaluate command line arguments and execute as command

SYNOPSIS

`eval` *argument ...*

DESCRIPTION

The shell built-in `eval` can be used to pass command line arguments to the `sh` or `ksh` shell for execution. The shell evaluates these arguments and then executes them as commands.

The specified command line arguments are thus evaluated twice by the shell:

- First, when the shell processes the `eval` command line.
- Second, when the processed command-line arguments are executed by the shell as commands. Each command line is processed by the shell before execution.

When do you need eval?

The shell processes each command line in a number of steps [see `sh(1)` , **COMMAND LINE PROCESSING**), interpreting specific metacharacters in each processing step. The original command line may be modified by the individual processing steps.

If, for example, the shell replaces a variable with its value or a command with its output, metacharacters may occur in the command line which may not be interpreted by the shell in subsequent processing steps. The shell built-in `eval` makes sure that metacharacters left by the previous step are interpreted in the following step (see also **EXAMPLES**).

OPERANDS

argument

Any string delimited by blanks or tabs. The last argument must be terminated by a command separator.

You may specify any number of arguments with at least one blank or tab between them.

The specified arguments are executed by the shell as a command.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

Example 1

The following command line cannot be executed as intended unless `eval` is used:

```
$ cmd='date;who'
$ $cmd
date;who: not found
$ eval $cmd
Thu Mar 09 15:46:02 MET 1989
udo      tty04   Mar  9 09:20
```

To begin with, the shell replaces the `cmd` variable with its value. After the value has been substituted for the variable, the shell no longer recognizes the colon as a command separator. Calling `eval` causes the shell to interpret the semicolon as intended, since the `$cmd` argument is processed twice, i.e. rescanned by the shell.

Example 2

The shell script `evaltest` will be used to demonstrate how the `eval` command is processed by the shell.

The script file has the following contents:

```
set –x
loginname=max
for i in 1 2 3 4
do
  eval group$i=$loginname$i
  eval echo \$group$i
done
```

`set –x` causes the shell running the shell script to write the commands processed to standard error output before executing them.

On executing this shell script, you will receive the following output on the screen (without the line numbers):

```
   $ sh evaltest
1  loginname=max
2  + eval group1=max1
3  group1=max1
4  + eval echo $group1
5  + echo max1
6  max1
.
.
.
```

This example shows only what is output the first time the loop is run.

Line 2 shows how the first `eval` command line is processed:

here the shell has substituted the value `1` for `$i` and the value `max` for `$loginname`. `eval` must be invoked as the shell does not recognize the equals sign as an assignment operator the first time the loop is run. The shell executes an assignment only if the variable name (i.e. the string in front of the equals sign =) begins with a letter or the underscore character _ and contains letters, numbers and underscores only. `group$i` is an illegal variable name as it contains the dollar character.

Line 3 shows that the assignment is made:

When the command line arguments are interpreted by the shell a second time, the equals sign is interpreted as intended, since the argument preceding the = sign represents a legal name for a shell variable.

Line 4 demonstrates how the second `eval` command line is processed:

The shell has substituted the value `1` for the positional parameter `$i` and removed the escape character \ preceding the `$`. `eval` is required in this case to ensure that the shell replaces the `group1` variable by its value.

Line 5 shows how the `echo` command is processed. Here, the shell has substituted the value `max1` for `$group1`. Line 6 contains the output of the `echo` command.

NOTES

Some differences in behavior may occur when using `eval`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh`(1), `set`(1), `sh`(1).

User's Reference Manual

ex(1)

NAME

ex - command and display editor

SYNOPSIS

ex [*option ...*] [*file ...*]

DESCRIPTION

ex is a line-based text editor. ex offers you various operating modes:

- In the ex input mode you enter text directly.
- In the ex command mode you enter commands. Among other things, commands enable you to:
  - position the cursor
  - search for (and replace) text patterns with the aid of regular expressions
  - switch to another file
  - invoke a shell

You also have the option of switching from the line editor ex to the screen editor vi.

Layout of this description

The description of how to call ex on Reliant UNIX shell level is followed by the sections:

Functionality of ex

- Operating modes of ex
- Saving the editor buffer and quitting ex
- Presetting ex
- Current and alternate files
- Regular expressions
- Replacement strings
- Buffers
- Error and signal handling

ex commands

- Addressing
- Parameters
- Commands

ex options

OPTIONS

−s (silent) All interactive output of the editor is suppressed. The −s option must be set when ex is to read its commands from a command script.

The −s option has replaced the old − option.

−t *tag*

(tag) ex loads the file containing the *tag* for editing and positions the tag definition under the cursor at the center of the screen. The tags file containing the search strings for the definitions must be in the same directory. This option is typically used by C programmers to quickly locate the definition of a function or macro on calling the editor. The tags file needed for this purpose must first be built with the ctags(1) command.

−r *file*

(recover) Restores the ex session in which you were editing *file* if ex or the system crashed during your session.

Changes that are only in the editor buffer are not copied to the edited file after a system or editor crash. Reliant UNIX does, however, try to back up the buffer contents by creating a copy of it, if possible. *file* is recovered, including any changes made prior to the crash, and is retrieved into the ex buffer. You can now continue editing the file or write the changes to another file.

−L Lists the names of all files saved following an editor or system crash.

−R (readonly) The named *file* is opened in readonly mode. This mode prevents the original *file* from being accidentally overwritten, but allows you to write the buffer contents to a file with another name.

Caution:

The file can still be overwritten with w! in the ex command mode.

−v (vi) Invokes the vi editor. A detailed description of vi is provided in vi(1) .

−c *command*

Positions the editor at a specific line of the file to be edited or executes an ex command when ex is invoked. If you position the editor at a line, the selected line will be displayed in the center of the screen. By contrast, if you have an ex command executed, the editor will be positioned at the last line of the file after executing the command, unless the ex command you specify (e.g. search) positions the editor itself.

command not specified:

The ex editor is positioned at the end of the file.

command specified:

command may be specified as a line number (*n*), a search command (/*pattern*), or any other ex command (see **EX COMMANDS**). The specified command is executed on invocation of ex:

n *n* must be an integer. ex is positioned to the *n*th line of the file.

/*pattern*

ex is positioned to the first line containing *pattern*. If the *pattern* you specify includes metacharacters, you must escape them to protect them from being interpreted by the shell.

'*ex-command*'
"*ex-command*"

ex-command can be any ex command. The ex command must be quoted (using single or double quotes) in order to prevent it from being interpreted by the shell. If the editor is not already positioned as a result of an ex command, ex will be positioned to the last line of the file.

Example:

When ex is called with:

```
ex -c /Tuesday appointments
```

the file appointments is opened, and the cursor is positioned at the first line that contains the word Tuesday [see vi(1)].

−x Encryption option: ex executes an X command and prompts you for a key. This key is used to encrypt and decrypt text using the algorithm of the crypt command. ex checks whether or not the text to be read is encrypted. The temporary buffer file is likewise encrypted, using a transformed version of the key you enter here [see crypt(1)].

−C Encryption option: the same as the −x option, except that ex executes the C command. The C command is like the X command, except that all text read in is assumed to have been encrypted [see crypt(1)].

file Name of the file that you wish to edit. If you specify a number of files, these files will be processed in the order in which they are listed. The ex command n can be used to switch to the next file.

file not specified:

ex opens an empty buffer into which you can write your text. The contents of this buffer are not assigned a name until you write them back (with w) to a file or assign a name explicitly using the f command.

Caution:

If your file contains null characters, ex will discard them when reading in the file. No null characters appear in files written by ex.

FUNCTIONALITY OF EX

ex always works on a buffer, not on files. When you start an ex session, a working copy of the file you are editing is read into the editor buffer. All the editing changes you make are initially performed in this buffer and are not saved until you use the w (write) command to copy the buffer contents back into the original file.

Once you have done this, you can quit ex using the q command.

If you wish to exit the ex editor without writing back the changes to the original file, you must enter q!.

Null ASCII characters [see ascii(5)] are discarded in input files and thus cannot appear in output files.

Operating modes of ex

ex offers you two file editing modes:

- the ex command mode, and
- the ex input mode.

You can also switch from ex to the vi editor, which provides you with additional operating modes (for more details see vi(1) , **Operating modes of vi**).

When ex is invoked, it comes up in command mode, as indicated by the prompt : displayed on the screen. The commands a (append), i (insert), and c (change) can be used to switch to the input mode, which enables you to add or modify text in the buffer (see **EX COMMANDS**).

In input mode, all subsequent input characters, including various non-printing characters, are written into the buffer. Commands are not interpreted as such. You can exit from input mode by entering a period (.) in the first column and then pressing RETURN.

Saving the editor buffer and quitting ex

ex must be in command mode to save the editor buffer or to quit the ex editor.

Use the following command to save the contents of your editor buffer to the file edited or that specified:

w [*file*]
: (write) The contents of the editor buffer are saved to the file specified.

  file not specified:

  The contents of the editor buffer are written back to the edited file.

You can quit ex using one of the following options:

q
: (quit) Quit ex. This works only if the editor buffer has not been modified at all or if the modified editor buffer has been saved to a file.

q!
: (quit) Quit ex. Unsaved modifications made to the editor buffer are not saved.

x
: Quit ex and write the modified editor buffer to the edited file.

wq [*file*]
: (write and quit) Quit ex and write the modified editor buffer to the *file* specified.

  file not specified:

  The contents of the editor buffer are written to the edited file.

  Caution:

  ex does not issue a warning if you place text in named buffers and do not save it before exiting the editor. The text is irretrievably discarded.

Presetting ex

The `ex` editor can, to some extent, be adapted to suit your requirements and working habits. This entails using the `ex` command `se` (set), which allows you to set or change specific options (see **EX OPTIONS**). The changes you make remain in effect for the current session. Details on how to change default settings permanently can be found in the `vi(1)` command description (see section **Presetting the vi editor**).

Current and alternate file

The file (or its copy) being edited by `ex` at any given moment is referred to as the current file. The alternate file is the file most recently mentioned in an editing command. If you switch to the alternate file, the file you switch from becomes the new alternate file. When specifying files, you can use the `%` character for the current file and the `#` character for the alternate file. These characters are replaced in file names by the names of the current file and the alternate file, respectively.

For example, the command:

```
w %.bak
```

backs up the current file to a file with the same name but with the extension `.bak`.

Regular expressions

The interpretation of metacharacters in regular expressions depends on whether or not the `magic` option has been set for the `ex` editor (cf. **EX OPTIONS**).

When `magic` is set:

character

An ordinary character stands for itself. The following are metacharacters, **not** ordinary characters:

`^`	(caret) at the beginning of a pattern
`$`	(dollar sign) at the end of a pattern
`*`	(asterisk) anywhere except at the beginning of a pattern
`.`	(period) at any position in a pattern
`[`	(left square bracket) at any position in a pattern
`~`	(tilde) at any position in a pattern

Metacharacters have special meanings and must be escaped with a backslash \ if they are to lose their special meanings.

`^` `^` at the beginning of a pattern stands for the beginning of the line.

`$` `$` at the end of a pattern stands for the end of the line.

`.` stands for any one character.

`\<`
`\>`

`\<` matches the beginning of a word. The word must begin with a letter, digit, or underscore and be preceded by the beginning of the line or a character other than the above. `\>` matches the end of a word.

`[`*string*`]`

Any character contained in *string*, where *string* is not a sequence of blank characters. The following characters have special meanings within *string*:

- a dash – between a pair of characters defines a range, e.g. `[a-z]` matches `a`, `z`, and all the characters which come between them in the ASCII collating sequence.
- if a caret `^` is the first character in *string*, it causes the construct to match any characters not included in *string*.

If these characters are escaped by a backslash, they lose their special meanings.

`*` Zero, one or more occurrences of the regular expression preceding the `*` character.

`~` Matches the replacement string used in the last `s` command (substitute, see **EX COMMANDS**).

\(*pattern*\)
A regular expression *pattern* can be enclosed in parentheses escaped by backslashes. This serves only to identify them for substitution actions (see **Replacement strings**).

rs A sequence of two regular expressions *r* and *s* is a regular expression in itself. *rs* matches all strings which consist of a string matching *r* followed by a string matching *s*.

When `nomagic` is set:

If `nomagic` has been set, the only characters that have special meanings are:

- caret ^ at the beginning of a pattern
- dollar `$` at the end of a pattern, and
- backslash \.

The following characters only have a special meaning if they have been escaped by a preceding backslash \:

- period .
- asterisk `*`
- left square bracket `[`
- tilde `~`

Replacement strings

The ampersand character `&` (with a preceding backslash, i.e. `\&`, if `nomagic` is set) in the replacement string stands for the text matched by the pattern, i.e. the text to be replaced.

The tilde `~` (`\~` if `nomagic` is set) is replaced by the replacement string used in the previous `s` (substitute) command.

The sequence *n*, where *n* is an integer, is replaced by the text that is matched by the pattern enclosed in the *n*th set of parentheses \(...\).

If the replacement string includes the sequence `\u` or `\l`, the character that immediately follows `\u` (upper) or `\l` (lower) in the replacement string will be converted to uppercase (for `u`) or lowercase (for `l`) if this character is a letter. The sequence `\U` or `\L` converts into uppercase or lowercase all letters until the end of the replacement string or until the sequence `\E` or `\e` is encountered.

Buffers

`ex` uses one unnamed and 26 named buffers, named `a` through `z`.

You can use these buffers to copy text with `ya` (yank), to save deleted text with `d` (delete), and to subsequently retrieve this text with `pu` (put).

The `d`, `pu`, and `ya` commands use the unnamed buffer unless you explicitly name a buffer. In other words, if you do not specify a buffer for a delete operation, for example, the material you delete will be saved in the unnamed buffer.

You can save blocks of text in up to 26 buffers. The buffers are designated in either lowercase or uppercase letters from a-z (or A-Z). If you use lowercase letters, the old contents of the buffer will be overwritten by the new text, i.e. the old contents are deleted. If you use uppercase letters, the old buffer contents are not overwritten; the new text is appended to the old one instead.

Error and signal handling

When an error occurs during an `ex` session, `ex` sends the BEL character [acoustic signal; see `ascii(5)`] to the terminal and prints an error message.

If an interrupt signal is received, `ex` returns to command mode, allowing you to enter a new `ex` command.

If the editor input is from a file, `ex` exits at the interrupt signal.

EX COMMANDS

Unlike most `vi` commands, which are interpreted and executed by `vi` immediately, `ex` commands must be "dispatched" by pressing the RETURN key.

Command lines which start with double quotes " are ignored. This means that you can use lines of this kind to insert comment lines in a command script.

Addressing

An address is used to specify a particular line. Some `ex` commands expect one or more addresses as input. The specified line or line range (an inclusive range of all lines from the first address to the second) is then processed by the `ex` command.

For `ex`, there is at all times a current line. This is explicitly addressed by a period (.). As a rule, the current line is the line last processed by a command or the line to which the editor has been explicitly positioned (e.g. by a search command).

Addresses must be separated from one another by a comma or a semicolon. Such address lists are evaluated by `ex` from left to right.

- When a semicolon (;) is used as a separator, the current line is set to the value of the first address before the second address is interpreted.
- When a comma (,) is the separator, all addresses are calculated relative to the current line. The current line is not changed until the command is executed.

If more addresses are given than the command requires, then all but the last address (when a line is expected) or the last two addresses (if a range is expected) are ignored. When a command requires two addresses, the first line that is addressed must precede the second one in the buffer. If you do not specify any address (null address), `ex` will use the current line by default.

`.` Current line of the buffer.

`$` Last line of the buffer.

`n` *n*th line in the buffer. The lines are numbered sequentially from 1.

`'`*x* The line marked with the letter *x*, where *x* must be a lowercase letter (see **EX COMMANDS**, `ma` or `k`). `'`*x* is the address of the marked line. Before `ex` executes a movement command, the current line is marked. You can return to it with "(2 single quotes).

/RE/ A simple regular expression enclosed in /.../ [see `expressions(5)`] addresses the first line which contains a character string which matches the regular expression, searching forward from the current line. If necessary, `ex` wraps around from the end of the buffer to its start to continue the search (see **EX OPTIONS**, `wrapscan`). If the line matching *RE* is only to be printed, the second / may be omitted.

RE not specified:

The previous specified pattern is used.

`?`*RE*`?` As *RE*, but searching backward.

+n
–n

If an address starts with a plus or minus sign followed by a decimal number *n* (optional), the line that lies *n* lines after (+) or before (–) the current line will be addressed. Thus, .+3, +3 and +++ have the same effect.

adr+
adr–

If an address ends with a plus or minus sign, the line that is located one line after (+) or before (–) the line identified by the address is referenced. A single – refers to the line before the current line. The + or – signs at the end of an address have a cumulative effect; the address ++ thus addresses the second line after the current line, 3++ addresses line 5, and ++ and .+2 are identical.

`%` The `%` (percent) sign stands for the address pair `1,$`, i.e. the entire buffer.

Parameters

The following parameters are used with `ex` commands:

line = *address*
: A single line address, which you can specify in any of the forms described earlier in the **Addressing** section.

line not specified:

The default value (the period `.`) applies. The period stands for the current line.

range = *address*`,`*address*
range = *address*`;`*address*
: A pair of line addresses that defines an inclusive range of lines. The addresses can be separated by a comma or a semicolon. A *range* should not be specified together with a number *n*, since the *last* address of the specified range will then be interpreted as the *first* address of a range of *n* lines that begins with the last address. In other words, you would end up addressing a range that comes after the one intended. This has been taken into account in the `ex` command syntax.

range not specified:

The default value `.,.` applies.

`.,.` represents only the current line.

n
: A positive integer. In *n* you specify the number of lines to be processed.

n not specified:

The default value is 1.

flag
: One or a combination of the `ex` commands `#` (see `nu`), `p`, and `l`. These commands display a line in a specific format and are executed after the preceding `ex` command completes.

The blanks need not always be specified. However, in the interest of clarity, they have not been indicated as optional here.

Any number of plus or minus characters may be combined with these parameters.

Commands

Overview of ex commands and their abbreviations

`CTRL-D`	`(scroll)`	display *n* lines, where *n*=`$scroll` (see options)
`ab`	`abbrev`	define abbreviations
`a`	`append`	add text after specified line
`ar`	`args`	display argument list of command line
`c`	`change`	replace text line
`C`	`encryption`	encrypt text
`co`	`copy`	copy text range
`d`	`delete`	delete text range
`e`	`edit`	change edited file
`f`	`file`	display current file name
`g`	`global`	execute command for all lines matching /*RE*/
`i`	`insert`	insert text lines
`j`	`join`	join two lines
`l`	`list`	output text with non-printing characters

map	---	define macros
k	mark	mark lines
m	move	move line range
ma	mark	mark lines
n	next	edit next file
nu	number	number and output lines

#	number	number and output lines
pre	preserve	preserve contents of buffer
p	print	print range of text
pu	put	put lines back in buffer
q	quit	quit the editor
r	read	read copy of a file into the buffer
rec	recover	reconstruct file after system crash
rew	rewind	rewind argument list (cf. ar)
se	set	display and set options
sh	shell	call shell
so	source	read command from file
s	substitute	search and replace
unab	unabbrev	cancel word from abbreviation list made with ab
u	undo	undo effect of previous command
unm	unmap	remove macro definition made with map
v	---	like g, but for all lines that do not match /*RE*/
ve	version	output version number of the editor
vi	visual	switch to vi
w	write	write buffer contents to file
x	exit	quit editor after saving buffer content
X	encryption	encrypt text
ya	yank	copy lines to buffer
z	(window)	output specific text range
!	(escape)	execute shell command
& s	(resubst)	repeat the last substitute command
<	(lshift)	shift text range to left
>	(rshift)	shift text range to right
=	(line no)	display the line number

No command specified

If you specify only a *line* or a *range*, the command p (print) is automatically assumed and executed. If a null line is entered, the next line is printed (equivalent to .+1p)

<CTRL-D>

(ASCII EOT) prints the next *n* lines, where *n* is the value of the `ex` option `scroll`.

`ab` word text
(abbreviate) This command is only effective in `vi` mode! If you enter *word* in the `vi` input mode as a complete word (i.e. not as a word fragment), it is replaced by the string *text*. If *text* is to include special characters, e.g. RETURN, these characters must be preceded by CTRL-V.

line `a`
(append) Causes `ex` to enter *input* mode, placing the input text after the specified line. Use a *line* value of 0 to put a line at the beginning of the buffer. The last line of new input becomes the new current line. If there is no new input, the target *line* becomes the current line.

Input is terminated by a period (.) entered in column 1.

`ar` (arguments) Prints the argument list from the command line of the `ex` call, enclosed in square brackets [...].

line `c` *n*
range `c`
(change) `ex` switches to *input* mode. *n* lines or the lines in *range* are replaced by the lines you enter next.

The last input line becomes the current line. If no text lines are entered, the specified range is deleted, i.e. `c` has the same effect as the `d` (delete) command. Input is terminated by a period (.) entered in column 1.

`C` Encryption command: the `C` command is like the `X` command, except that all text read in with the `e` and `r` commands is assumed to have been encrypted (see option `–C`).

range `co` *line flags*
(copy) Copies the lines specified by *range* to after *line*. If *line* is assigned a value of 0, the *range* of lines (i.e. text) will be copied to the beginning of the buffer.

line `d` *buffer n*
range `d` *buffer*
(delete) The lines in *range*, or *n* lines from the given *line*, are deleted from the buffer. If a named *buffer* is specified, the deleted text is saved in it. You can use the `ex` command `pu` to retrieve the contents of this buffer if you wish. The first line following the deleted material becomes the current line, unless the deleted lines were at the end of the buffer. In that case, the last line in the buffer becomes the current line.

`e` [+*line*] *file*
(edit) The named *file* is edited. If the current buffer has been modified since the last `w` (write) command, `ex` warns you and aborts the command. You can, however, force the execution of the `e` command by appending an exclamation mark to the `e`: `e!` file. The current line is the last line of the buffer; however, if this command is called from `vi`, the current line becomes the first line of the buffer.

+*line* The specified *line* becomes the current line. *line* can be given as:

- aline number *n*
- the dollar symbol $ (for end of file) – a regular expression in the form /*RE* or ?*RE*.

`f` [*name*]
(file) Prints the current file name and other information, e.g. the number of lines and the position of the current line.

name specified:

name becomes the current file name

range `g/`*RE*`/`*cmds*
(global) First marks all lines within the given *range* that match the pattern specified in *RE*; then applies the given *cmds* to the each of marked lines in turn, with each marked line becoming the current line while it is being edited.

The commands given in *cmds* may be specified on separate lines, provided newline characters are escaped with a backslash. If *cmds* are omitted, all lines will be printed. The `a(ppend)`, `c(hange)`, and `i(nsert)` commands are allowed; the terminating period at the end of *cmds* may be omitted. The `visual` command is also permitted, and takes input from the display terminal.

The following commands are not permitted in *cmds*: `global` and `undo`.

The following edit options are not allowed in *cmds*: `autoprint`, `autoindent` and `report`.

Example:

```
1,$g/^\*/s//+/
```

All occurrences of `*` in the first column `^` of line 1 and of all lines through to the end of the file (`$`) are replaced by a plus character (+) by the substitute command (`s`).

line `i`

(insert) `ex` enters input mode; the input text is placed before the specified line. The last line inserted becomes the current line, but if no text is inserted, the line before the target *line* becomes the current line. Input is terminated by a period (.) entered in column 1.

line `j` *n flag*
range `j` *flag*

(join) Joins the lines in *range*, or *n* lines from *line*, into one line. The command inserts at least one blank between two previously split lines; two blanks are provided if a line ends with a period. If a following line begins with a right parenthesis, no blanks are inserted. Extra white space at the start of a line is discarded.

Appending an exclamation mark to the `j` command (`j!`) produces a simpler join with no white space processing.

line `l` *n flag*

range `l` *flag*

(list) Displays the specified lines. Tab characters are replaced by `^I`, and the end of each line is marked with a trailing `$`. The only useful *flag* is `#`, when the lines are to be preceded by their line numbers. The last line printed becomes the current line.

`map` *x commands*

`map!` *x text*

Only works with `vi`!

Format 1: Macro definitions for `vi` command mode

This format of the `map` command is used to define macros for use in the `vi` command mode. The first argument *x* is a single character, or the sequence `#`*n*, where *n* is a digit that refers to function key *n*. When this character or function key is typed in `vi` command mode, `vi` responds as if the corresponding *commands* were entered. The specified *commands* are interpreted as a sequence of `vi` commands. If special characters, white space, or newline characters are to be used in the *commands*, they must be escaped with CTRL-V.

Example 1:

You want to define a macro which replaces an `*` occurring at the beginning of a line (column 1) by a blank throughout the entire file. The name of this macro is to be `*`. Enter the following in your `.exrc` file (see section **Presetting ex**) so that you can call the macro when required:

```
:map * :1,$s/^\*/ /
```

Example 2:

You now want to define a macro and bind it to function key <F1>. This macro is to search for the next line that begins with a particular regular expression *RE* and then automatically delete this line.

To do this, you enter (from `vi` command mode):

`:map` CTRL-V <F1> `/^rA` CTRL-V RETURN `dd` RETURN

Format 2: Macro definitions for `vi` input mode

This format of the `map` command defines macros for use in `vi` input mode (cf. **Format 1**). Every *x* entered in `vi` input mode is replaced by *text*. (In the `ab` command, by contrast, *x* is replaced only when alone).

If *x* is not to be replaced by *text*, it must be escaped by preceding it with CTRL-V.

Formats 1 and **2**:

Macro definitions can be canceled with:

`unmap x` or

`unmap! x`

line `ma` *x*
line `k` *x*

(mark) Marks the specified *line* with the character *x*, which must be a single lowercase letter. The *x* must be preceded by a blank or tab. The current line position is not affected.

range `m` *line*

(move) Moves the specified lines (*range*) to after the target line. The first of the moved lines becomes the current line.

`n` [*file* ...]

file not specified:

(next) The next file from the command line argument list is edited (cf. `f` and `ar`). If the current buffer has been modified since the last `w` (write) command, `ex` warns you and aborts the command. You can, however, force the execution of the `n` command by entering an exclamation mark after the `n`: `n!`.

file specified:

The previous argument list is replaced by the specified file(s), and the first specified file is edited.

line `nu` *n flag*
range `nu` *flag*
line `#` *n flag*
range `#` *flag*

(number) Prints the lines, each preceded by its line number. The only useful flag in this case is `l`. The last line printed becomes the current line.

`pre` (preserve) The current editor buffer is saved as though the system had just crashed. The `pre` command is for use in emergencies, e.g. when a `w` (write) does not work because the disk is full, and the buffer contents cannot be saved in any other way.

line `p` *n*
range `p`

(print) Prints the specified lines, with non-printing characters printed as control characters in the form ^*x*; DEL is represented as `^?`. The last line printed becomes the current line.

line `pu` *buffer*

(put) Lines deleted with `d` (delete) or copied to a buffer with `y` (yank) are put back in after the specified *line*. *buffer* represents a buffer name from `a` to `z`.

buffer not specified:

The text in the unnamed buffer is retrieved.

`q` (quit) Quits the editor. If the buffer has been modified since the last `w` (write) command, a warning is printed and the `q` command fails. You can, however, force the execution of the `q` command by appending an exclamation mark to the `q`: `q!`. All changes not saved with `w` will then be discarded.

line `r` [*file*]

(read) Places a copy of the specified *file* in the buffer after the target *line*. Use a value of 0 to place text at the beginning of the buffer. If there is no current file at this point (e.g. when `ex` is called without a file name), then *file* becomes the current file. The last line read becomes the current line; in `vi` mode, the first line read becomes the current line.

file not specified:

The current file is read.

If *file* is given as `!`*unixcmd*, then *unixcmd* is interpreted as a Reliant UNIX command and passed to the shell. The output of the Reliant UNIX command is then read in to the buffer.

`rec` file
: Recovers *file* from the save area after an editor or system crash.

`rew`
: (rewind) The argument list (see `ar`) is rewound, and the first file in the list is edited. If the buffer has been modified since the last `w` (write) command, a warning is printed and the `rew` command is not executed. You can, however, force the execution of the `rew` command by appending an exclamation mark to the `rew` command: `rew!`. All changes not saved with `w` are then discarded.

`se` [*parameter*]
: (set) The `set` command can be used to display or set options. There are two types of options: those that have "Boolean" (i.e. on or off) values and those that do not. One example of a Boolean parameter is the `number` option. If this option is set, line numbers are displayed; when `nonumber` is set, no line numbers are displayed. By contrast, the `report` option is non-Boolean. The value of this option (default = 5) defines the number of lines that must be changed by a command before `ex` and `vi` issue a message.

parameter
`all`
: The values of all options are displayed.

option`?`
: The current value of the specified option is displayed.

option
: Only for options with Boolean values: The option is set.

`no`*option*
: Only for options with Boolean values: The option is not set.

option`=`*value*
: Only for options with non-Boolean values: The option is assigned the specified *value*.

parameter not specified:

`set` displays user-defined `ex` options, i.e. options that deviate from default settings.

More detailed information on `ex` options is provided in the corresponding section below.

`sh`
: If the `SHELL` variable is set, the shell defined in that variable is invoked; otherwise, the Bourne shell `sh`. When you exit the shell, you will be returned to the editing session at the point you left off.

`so` *file*
: Reads and executes commands from the specified *file*. Such `so` commands may be nested.

line `s/`*RE*`/`*repl*`/`*options* *n* *flags*
range `s/`*RE*`/`*repl*`/`*options* *flags*
: (substitute) The first occurrence of a pattern that matches *RE* (regular expression) in each line of the given range is replaced by the string *repl* (see **Regular expressions** and **Replacement strings**). The range is either given in *range* or specified as *n* lines from the specified *line*.

options
`g`
: (global) *All* strings that match *RE* in the line are substituted.

`c`
: (confirm) Before each substitution, the line is output with the pattern to be replaced shown in the line below it, marked by a `^` character. Entering `y` causes the substitution to be made; any other input causes the command to abort. The last line that contains a substitution becomes the current line.

`una` *word*
: (unabbreviate) *word* is deleted from the list of abbreviations set up by `ab`.

`u`
: (undo) Reverses the changes made by the previous editing command. `g` (global) and `vi` (visual) are

considered single commands in this case. Commands which affect the external environment, such as `w` (write), `e` (edit), and `n` (next), cannot be undone.

An `undo` can itself be reversed with a follow-up `u`.

All markers for lines changed and subsequently restored are lost with `u`.

`unm` *x*
`unm!` *x*
(unmap) Removes the macro definition created by `map` for *x*.

range `v/`*RE*`/`*cmds*
(vice versa) This is the same as the `global` command, except that *cmds* are applied to all lines that do not match the pattern *RE*.

`ve` Prints the current version of the editor.

line `vi` [*type*] *n*
Enters `vi` mode at the specified *line*. The *type* is optional and may be given as a minus sign (–) or a period (.), as in the `z` command, to specify the position of the specified *line* on the screen window. The default is to place the line at the top of the screen window. *n* specifies an initial window size; the default is the value of the `ex` option `window`. The command `Q` exits `vi` mode and returns you to `ex`. For further information, see `vi(1)` .

[*range*] `w`[`!`] [*file*]

[*range*] `wq`[`!`] [*file*]

Format 1: Write to a file

(write) The specified *range* is written to the named *file*, and the number of lines and characters written is printed.

If an *alternate* file is specified, and the file exists, the command will fail in order to prevent the alternate file from being accidentally overwritten. The execution of `w` can, however, be forced by appending an exclamation mark to the command: `w!`.

To append to the *file*, use the form `w>>`. If the *file* does not exist, an error message is issued.

If !*sinixcmd* is specified instead of *file*, then *sinixcmd* is interpreted as a Reliant UNIX command. The command interpreter is invoked, and the specified *range* is passed as standard input to the command.

range not specified:

The entire current file is written to *file*.

file not specified:

The current file is used by default. The `w` command cannot be executed if there is no current file and no *file* is specified.

Format 2: Write to a file and quit `ex`

(write and quit) The command `wq` is equivalent to a `w` followed by a `q`; `wq!` is equivalent to `w!` followed by `q`.

`x` (exit) Writes out the buffer if changes have been made since the last `w` and then (in any case) quits.

`X` Encryption command: you are prompted for a key, which is used in subsequent `e`, `r`, and `w` commands to decrypt and encrypt text using the algorithm of the `crypt(1)` command. A null key turns off encryption. `ex` verifies whether or not the text to be read (`e` and `r`) is encrypted. The temporary buffer file is likewise encrypted, using a transformed version of the key you enter here (see the `–x` option).

line `ya` *buffer n*
range `ya` *buffer*
Copies (yanks) the specified *range*, or *n* lines from the specified *line*, to the named *buffer*.

buffer not specified:

If no buffer is specified, the unnamed buffer is used.

line `z` [*type*] *n*
(window) Displays *n* lines from the area in which *line* is located.

n not specified:

n defaults to the value of the `window` variable.

type

type is an optional parameter and can be specified as a dash or a period; a – causes the *line* to be placed at the bottom of the screen, while a . causes the line to be placed in the middle of the display. The last line printed becomes the current line.

type not specified:

line is placed at the top of the displayed area.

Caution:

`ex` prints the number of logical rather than physical lines. More than a screen full of output may result if there are lines which are longer than the screen width.

`!` *command*

(escape) The remainder of the line after the exclamation mark is passed to the system command interpreter for execution. A warning is issued if the buffer has been changed since the last `w` (write) command. The specified command is then executed. If the *command output* has not been redirected, it is displayed on the screen, but does not change the file. The *command itself* may, however, modify the file, e.g. `!cat␣/etc/profile␣>>␣%`. A single `!` is printed when the command completes. The current line position is not affected.

Within the text of *command*, the percent sign `%` and the hash character `#` are expanded as file names (see **Current and alternate files**).

An exclamation mark in *command* is replaced with the text of the previous `!` command. Thus, `!!` means "repeat the preceding command". If any such expansion is done, the expanded line will be echoed.

range `!` *command*

(escape) In this form of the `!` command, the lines specified in *range* are passed to the *command* as standard input and replaced by the output of the *command*.

range not specified:

range is not replaced by the current line (. , .); the other form of the `!` command applies.

`"` Command lines that start with double quotes `"` are ignored. This means that you can use lines of this kind to insert comment lines in a command script.

line `&`*options n flags*
range`&`*options flags*
line `s`*options n flags*
range`s`*options flags*

(resubst) Repeats the previous `s` (substitute) command, as if `&` were replaced by the previous `s/`*RE*`/`*repl*`/`. The same effect is obtained by omitting the `/`*RE*`/`*repl*`/` string in an `s` command.

line `<` *n*
range `<`

(lshift) Shifts the defined line range (or the *n* lines that follow *line*) to the left by the number of spaces specified by the `shiftwidth` option. White space (blanks and tabs) is lost in the shift process; other characters are not affected. The last line changed becomes the current line.

line `>` *n*
range `>`

(rshift) Shifts the defined line range (or the *n* lines that follow *line*) to the right by the number of spaces specified by the `shiftwidth` option. White space (blanks and tabs) is not lost, but inserted as required.

line`=` (line number) Prints the line number of the specified *line*. The current line position is not affected.

line not specified:

The line number of the last line is displayed.

EX OPTIONS

`ex` provides a number of options with which you can govern the behavior of both the `ex` and (more importantly) the `vi` editors (see `vi(1)` , **Presetting the vi environment**).

All options have default settings; these settings are valid at the time `ex` or `vi` is invoked. Use the `ex` command `se` (set) if you wish to have all options displayed:

`set all` RETURN

The above command can also be used to change one or more options:

`set showmode scroll=15` RETURN

This command shows the `vi` input mode in the status line and sets the number of lines scrolled by the CTRL-D key to `15`.

The command:

`set` RETURN

displays all options set to non-default values.

There are two types of options: those that have "Boolean" (i.e. on or off) values and those that do not. One example of a Boolean parameter is the `showmode` option. When this option is not set, it is named `noshowmode`. The `scroll` option, by contrast, is an example of a non-Boolean parameter.

`autoindent`, `ai`
`noautoindent`, `noai` (default)
: If `autoindent` is set, each line in insert mode is aligned with the beginning of the previous line; tabs and blanks are inserted to produce this alignment. The starting indentation of a line is determined by the line it is appended after (`a`), the line it is inserted before (`i`), or the first line to be changed (`c`). Extra indentation can be provided as usual; succeeding lines will automatically be indented to the new alignment.

  Reduced indentation is achieved by typing CTRL-D to back up over the indentation by `shiftwidth` spaces. A caret ^ followed by a CTRL-D removes all indentation temporarily for the current line; a `0` (zero) followed by a CTRL-D removes all indentation.

`autoprint`, `ap`
`noautoprint`, `noap` (default)
: Prints the current line after each command that changes buffer text. `autoprint` is suppressed during global search and replace operations (see `g`, `s` and `v`).

`autowrite`, `aw`
`noautowrite`, `noaw` (default)
: Writes the buffer to the current file if the buffer has been modified, and an `e` (edit), `n` (next), `rew` (rewind) or `!` (escape) command has been given.

`beautify`, `bf`
`nobeautify`, `nobf` (default)
: Causes all control characters other than tab, newline and form feed to be discarded from the input text.

`directory=`*dir*
`dir=`*dir*
: The value of *dir* specifies the directory which is to hold the editor buffer. If the user does not have write permission for this directory, the editor quits.

`edcompatible`, `ed`
`noedcompatible`, `noed`
: Causes the presence of `g` and `c` suffixes on `s` (substitute) commands to be remembered for use as toggles.

`errorbells`, `eb`
`noerrorbells`, `noeb`
: Rings the terminal bell before displaying error messages in the last line (usually in reverse video).

`exrc`, `ex`

This option has been deactivated for security reasons. It normally causes `.exrc` files in the current directory to be evaluated.

`flash`, `fl` (default)
`noflash`, `nofl`
If the terminfo description contains an entry for flashing the screen to indicate an error, this option can be used to inform users when they have made an invalid entry.

`hardtabs=`*number*
`ht=`*number*
If the terminal has hardware tabs, this option can be used to set the number of spaces to which the tab stops are set (to optimize screen output).

`ignorecase`, `ic`
`noignorecase`, `noic` (default)
Treats all uppercase characters in the text as lowercase when a search is made. Uppercase characters in regular expressions are also mapped to lowercase, except in character class expressions.

`lisp`
`nolisp` (default)
Activates the `autoindent` option and modifies the `vi` commands left parenthesis (, right parenthesis), left brace {, right brace }, double left brackets `[[`, and double right brackets `]]` suitably.

`list`
`nolist` (default)
All printed lines will be displayed with tabs shown as `^I`, and the end of line marked by a `$`.

`magic` (default)
`nomagic`
Changes interpretation of characters in regular expressions and substitution replacement strings (see the relevant sections).

`mesg` (default)
`nomesg`
Grants/denies other users permission to write to the terminal (e.g. with the `write` command).

`modelines` (default)
`nomodelines`
The first and last five lines of a file that is read in are interpreted as editor commands and executed if they are in the form: `ex:`*command*`:`

`novice`
`nonovice` (default)
Defines the level of detail for error messages.

`number`, `nu`
`nonumber`, `nonu` (default)
Causes lines to be printed with line numbers.

`optimize`, `opt`
`nooptimize`, `noopt` (default)
Allows you to set the terminal with or without automatic carriage returns (to optimize output).

`paragraphs=`*string*
`para=`*string*
The value of this option is a string in which each successive pair of characters specifies the name of an `nroff` macro that begins a paragraph. A macro appears in the text in the form `.`*XX*, where the `.` is the first character in the line.

`prompt` (default)
`noprompt`
When `prompt` is set, the `:` prompt is displayed in command mode; otherwise, there is no command-mode prompt.

`readonly`

`noreadonly` (default)
: If `readonly` is set, the editor can only be used to read files, not to make changes to the text. If `noreadonly` is set, you can make changes to files.

`redraw` (default)
`noredraw`
: The editor simulates an intelligent terminal on a dumb terminal. Since this is likely to require a large amount of output to the terminal, it is useful only at high transmission speeds.

`remap` (default)
`noremap`
: If `remap` is set, macro translation allows for macros defined in terms of other macros; translation continues until the final product is obtained. Only one step of translation is performed if `noremap` is set.

`report=`*n*
: When the number of lines changed, deleted, or copied by the last command exceeds *n*, a message is issued in the status line.

`scroll=`*n*
: The value *n* specified for `scroll` determines the number of lines that the screen will scroll when a CTRL-D command is entered. It also controls the number of lines displayed by a `z` command (twice the value of `scroll`).

`sections=`*string*
`sect=`*string*
: The value assigned to this option is a string in which each successive pair of characters specifies the name of an `nroff` macro which begins a section. A macro appears in the text in the form `.`*XX*, where the `.` is the first character in the line.

`shiftwidth=`*n*
`sw=`*n*
: □
The value *n* specified for `sw` is used for the spacing of tab stops used by the tab key, the `autoindent` option, and by the `<` and `>` (shift) commands.

`showmatch`, `sm`
`noshowmatch`, `nosm` (default)
: When a right parenthesis `)` or right brace `}` is entered in `vi`, the matching left parenthesis `(` or left brace `{` is shown, provided the matching character is still on the screen.

`showmode`, `smd`
`noshowmode`, `nosmd` (default)
: When `smd` is set, an indication is provided in the status line as to whether `vi` is in `vi` input mode.

`slowopen`, `slow`
`noslowopen`, `noslow` (default)
: In `vi`, prevents screen updates during input to improve throughput on unintelligent terminals.

`tabstop=`*n*
`ts=`*n*
: □
n specifies the software tab spacing to be used by the editor when expanding tabs in the input file.

`taglength=`*number*
`tl=`*number*
: Defines how many characters are to be considered significant in searches with the `:`*tag* command. A value of 0 means that all characters are to be considered.

`tags=`*files*
: Names of tag files, separated by blanks.

`term=`*string*

string tells `vi` what type of terminal is being used. (The default is the value of the environment variable `TERM`.)

`terse`
`noterse` (default)
Setting `terse` causes shorter error messages to be issued.

`timeout`
`notimeout`
Sets/cancels a timeout period.

`ttytype=`*string*
string tells `vi` what type of terminal is being used. (The default is the value of the environment variable `TERM`.)

`warn` (default)
`nowarn`
When `warn` is set, the warning `No write since last change` is issued if a `:!` command is given before the most recent changes to the editor buffer have been saved with `w`.

`window=`*n*
The number of lines in a `vi` text window.

`wrapmargin=`*n*
`wm=`*n*

Only effective in `vi`. If *n*>0, a newline is automatically added to an input line at a word boundary such that lines end at least *n* spaces from the right margin of the terminal screen.

The setting of automatic line breaks can be deactivated by setting the value of *n* to 0 (default).

`wrapscan`, `ws` (default)
`nowrapscan`, `nows`
Setting `wrapscan` causes searches begun by the /.../ or ?...? commands to wrap around from the end of a buffer to the beginning (or vice-versa). When `nows` is set, the search runs through to the end or beginning of the buffer as appropriate and then terminates.

`writeany`, `wa`
`nowriteany`, `nowa` (default)
When `nowa` is set, a check is performed before a `w` command to determine whether the file exists. This is to prevent an existing file from being accidentally overwritten. You can override the overwrite protection with `w!`. Setting `wa` inhibits the normally provided overwrite protection.

ERROR MESSAGES

When an error occurs, `ex` sends the acoustic signal BEL [see `ascii(5)`] to the terminal and prints a message. If an interrupt signal is received, `ex` returns to command level, allowing you to enter a new `ex` command. If the editor input is from a file, `ex` exits at the interrupt signal.

If `ex` detects an internal error, it attempts to preserve the buffer if the most recent changes have not been saved. If you wish to access the backed up data, you must call `ex` with the `–r` option.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

The following example demonstrates and explains a number of `ex` commands:

```
$ ex figures                    – Call ex (file name "figures")
"figures" 4 lines, 19 characters
:set number             – Output line numbers
```

```
:1,$p                   – Output line 1 through to last line□
    1  one               – Original contents of "figures"□
    2  two□
    3  three□
    4  four□
:1 c 1                  – Modify line 1□
    1  ONE               – New line 1□
    2  .                 – Quit input mode□
:2,3 co 4               – Copy lines 2 to 3 after line 4□
    6  three□
:1,$ g/two/s//TWO/      – Replace "two" by "TWO" throughout□
 □

                        the file□
:2 a                    – Insert new line after line 2□
    3  THREE            – New line 3□
    4  .                – Quit input mode□
:4 i                    – Insert new line before line 4□
    4  FOUR             – New line 4□
    5  .                – Quit input mode□
:5,$ d a                – Delete line 5 and all lines to□
                        end of file□
    4  FOUR□
:1,$p                   – Output line 1 through to last line□
    1  ONE□
    2  TWO□
    3  THREE□
    4  FOUR□
:wq                     – Save the contents of the buffer□
 □

                        and quit ex□
"figures" 4 lines, 19 characters
```

ENVIRONMENT VARIABLES

`TERM` The type of data terminal used must be defined in the environment variable `TERM`.

`EXINIT`
: Environment variable with default values for `ex` and `vi`. These defaults always apply.

FILES

`$HOME/.exrc`
: File containing default values for `ex` and `vi`. These defaults are overridden if conflicting settings have been assigned to `EXINIT`.

SEE ALSO

`ed(1)`, `edit(1)`, `vi(1)`.

User's Reference Manual

exec(1)

NAME

`exec` - overlay current shell (execute)

SYNOPSIS

Format 1

`exec` *program* [*redirection*]

Format 2

`exec` *redirection*

DESCRIPTION

The shell built-in `exec` performs two functions:

- It overlays the current shell with another program (**Format 1**). When this program starts, the current shell is terminated. No new process is created, as is evident from the fact that the process ID does not change (see *Example 2*).

  If you enter `exec` interactively, when the specified program exits you return to the parent shell of the previous shell; if your previous shell was a login shell, your session terminates.

  If `exec` is called from a shell script, the script terminates. Commands that follow an `exec` call in the script will never be executed.
- It can be used to redirect the standard input or the standard output of the shell to a file (**Format 2**). All commands entered after `exec` has been executed will read from or write to this file until you terminate the current shell.

The Korn shell `ksh` has additional redirection options [see `ksh(1)`].

Format 1: Replacing the shell with another program

`exec` *program* [*redirection*]

program

Any command, program or shell script, but not another shell built-in. You will need execute permission for the associated file.

The current shell terminates, and *program* is executed instead. On completion of the specified *program*, control reverts to the old shell's parent, or the welcome screen is displayed of the old shell was a login shell.

redirection

If the specified program reads from standard input or writes to standard output, a file can be assigned for the input or output instead.

redirection can be specified as follows:

>*file*

The standard output of the specified program is redirected to *file*. Any data previously in *file* will be deleted.

>>*file*

The standard output of the program specified is redirected to *file*. Data already contained in *file* is not deleted; program output is appended to it instead.

2>*file*

The standard error output of the specified program is redirected to *file*.

<*file*

The standard input of the specified program is redirected to *file*, i.e. the program reads its input from this file.

Format 2: Redirecting the shell's standard input/standard output

`exec` *redirection*

redirection

Commands that read from standard input or write to standard output are assigned a file for their input/output. The redirection applies to all commands that the current shell executes after `exec`.

redirection can be specified as follows:

`>`*file*

All commands executed by the current shell write their output to the specified file sequentially. Data previously in *file* is deleted.

The output from all commands can be collected in this way.

If you have entered `exec` interactively, the redirection can only be canceled by:

- pressing CTRL-D. This terminates the current shell.
- entering `exec >/dev/tty`. This redirects the standard output back to the screen.

If `exec` is called from a shell script, the redirection will only apply to commands that follow the `exec` call in the script. Redirection can be canceled within the script by including the command `exec >/dev/tty` at the appropriate position in the script. This will redirect the standard output of all subsequent commands back to the screen.

`>>`*file*

All commands executed by the current shell write their output to the specified file sequentially. Data already contained in *file* is not deleted; program output is appended to it instead.

`2>`*file*

Standard error is redirected to *file* for all commands executed by the current shell.

`<`*file*

If you specify `exec` interactively, the current shell will read the commands to be executed from the specified file. The shell exits after the last command.

If this command appears in a shell script, all commands that follow the `exec` call in the script read their input from the specified file. Each read operation modifies the position of the read pointer in this file. If the read pointer is set to the end of the file (EOF), all following commands will receive no input.

The command `exec </dev/tty` can be used to redirect the standard input back to the keyboard for all subsequent commands.

file If you have entered `exec` interactively, *file* must be the name of a shell script. You will only need read permission for this shell script.

When `exec` is called from a shell script, *file* designates the name of the file from which all subsequent commands are to obtain their input.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

What happens when you make the following input?

```
$ exec date
```

First, the `exec` command terminates the shell and then causes the `date` command to be executed. The

current date is displayed on the screen.

If you entered the `exec` command from your login shell, the welcome screen will then be displayed. You will now need to log into the system again.

If you entered the command from a subshell, control will be returned to the parent shell.

Example 2

As indicated earlier, the process ID does not change when the current shell is replaced by another program. This concept is demonstrated with reference to the following files:

- Contents of the file `proc1`:

```
echo The process ID of proc1 is: $$ 
sh proc2
```

- Contents of the file `proc2`:

```
echo The process ID of proc2 is: $$ 
exec proc3
```

- The file `proc3`, which must be executable, contains the following:

```
echo The process ID of proc3 is: $$
```

The shell script `proc1` is now initiated:

```
$ sh proc1
The process ID of proc1 is: 2755
The process ID of proc2 is: 2760
The process ID of proc3 is: 2760
```

Since `proc3` is called from `proc2` with `exec`, both shell scripts run under the same process ID. To be exact, the shell that executes `proc2` is replaced by the shell that executes `proc3`.

Example 3

The `exec` command is used in the `exctest` shell script to redirect the standard input to the file `/etc/group` for all following commands. The contents of the `exctest` file are given below:

```
: Invocation with sh exctest
exec </etc/group
read lines
echo $lines
echo
cat
```

The shell script `exctest` is now initiated:

```
$ sh exctest
root::0:root

daemon::1:daemon
sys::2:sys:
bin::3:bin,admin
uucp::4:
.
.
.
```

The shell built-in `read` assigns the first line of the `/etc/group` file to the variable `line`. The `echo $line` command outputs the content of this variable.

The `cat` command also reads its input from the `/etc/group` file. Since the read pointer is now set to the second line of this file, `cat` displays the file's contents from the second line onward.

The read pointer is now set to EOF. Any further commands after the `cat` call would consequently receive the null string as input.

NOTES

Some differences in behavior may occur when using `exec`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh`(1), `sh`(1), `exec`(2).

User's Reference Manual

exit(1)

NAME

`exit -` terminate a shell script

SYNOPSIS

`exit` [*exit_value*]

DESCRIPTION

The shell built-in `exit` is used to terminate shell scripts. If you wish, you can define an exit status for the terminated script by following `exit` with a number.

Without `exit`, shell scripts terminate:

- when the last command has been executed, i.e. when the executing shell detects an end-of-file (EOF). The exit status is then the status of the last command executed.
- when the executing shell cannot find a command or detects a syntax error; the exit status is then non-zero.

If you call `exit` interactively, the shell in which you are currently working terminates.

OPERANDS

exit_value

Any number between 0 and 127. This number defines the exit status with which the shell script terminates.

The exit value can be checked with `echo $?`.

exit_value not specified:

The exit value of the command executed before the `exit` call is used.

EXAMPLES

Example 1

The `false` command can be implemented by means of the following shell script:

```
: Shell version of the false command
: Exit status always 1
exit 1
```

Example 2

The two shell scripts named `end` and `check` will be used to demonstrate how the exit status can be evaluated.

- The `end` script contains the following:

```
: Invocation with sh end file
if [ –f "$1" ]
then exit 2
elif [ –d "$1" ]
then exit 3
else exit 4
fi
```

- The `check` file is scripted as follows:

```
: Invocation with sh check file
sh end "$1"
case $? in
     2) echo Contents of file $1:; cat $1;;
     3) echo Contents of $1:; ls -l $1|pg;;
```

```
        4) echo Entering $1 is meaningless for this script!
   esac
```

The shell script `check` is now initiated:

```
$ sh check .profile
Contents of file .profile:
HOME=/usr1/rose/src
export HOME
.
.
.
```

The shell script `check` calls the script named `end`. This script returns an exit status of 2, since `.profile` is a normal file. The commands specified in the case statement under the matching pattern 2 are then executed.

The contents of the above two shell scripts cannot be simply combined into a single file, since the `exit` command terminates the script.

This shell script `endaction` shows how the exit status can be interpreted within a script:

```
: Invocation with sh endaction file
(if [ –f "$1" ]
   then exit 2
   elif [ –d "$1" ]
   then exit 3
   else exit 4
 fi)
 case $? in
      2) echo Contents of file $1:; cat $1;;
      3) echo Contents of $1:; ls –l $1|pg;;
      4) echo Entering $1 is meaningless for this script!
 esac
```

The parentheses around the if statement cause a subshell to be invoked. This subshell terminates with the exit status that is specified in the relevant `exit` command. Control is subsequently returned to the parent shell, i.e. the shell that processes the `endaction` script. This shell executes the remaining commands.

NOTES

Some differences in behavior may occur when using `exit`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`false`(1), `ksh`(1), `sh`(1), `exit`(2).

expand(1)

NAME

`expand` - convert tabs to spaces

SYNOPSIS

`/usr/ucb/expand` [`–t` *tablist*] [*file* ...]

DESCRIPTION

The `expand` command writes files or the standard input to the standard output with tab characters replaced by one or more blanks needed to pad the line to the next tab stop. All backspace characters are copied to the output and cause the column position count for tab stop calculations to be decremented. The column position count will not be decremented below zero.

OPTIONS

`–t` *tablist*

Specifies the tab stops. The argument *tablist* must consist of one or more numbers, separated by blanks or commas, in ascending order. A list separated by blanks must be enclosed in quotes. If only one number is specified, the tab stops will be set to *tablist* column positions instead of the default 8 column positions. If multiple numbers are given, the tabs will be set at the specified column positions.

Each tab stop position *N* must be an integer value greater than zero, and the specifications must be in ascending order. This means that tabbing from the start of the line of output to position *N* causes the next character output to be in the (*N*+1)th column position in that line.

If the `expand` command has to process a tab character at a position beyond the last position specified in a multiple tab stop list, the tab character is replaced by a blank in the output.

file The file whose tab characters are to be replaced by blanks.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of `expand`:

`LANG` Specifies a default value for the internationalization variables that are unset or null. If `LANG` is unset or null, the corresponding default value from the internationalized environment is used. If one of the internationalization variables contains an invalid setting, the command behaves as if none of the variables have been defined.

`LC_ALL`

If this variable has been assigned a value, i.e. it is not a null string, this value overrides the values of all the other internationalization variables.

`LC_CTYPE`

Determines the internationalized environment for the interpretation of byte sequences (e.g. single-byte characters as opposed to multi-byte characters in arguments and input files). The internationalized environment for the processing of tab characters and blanks and for the specification of the width in column positions each character would occupy on a constant-width output device.

`LC_MESSAGES`

Determines the format and contents of error messages.

`NLSPATH`

Determines the position of message catalogs for the processing of `LC_MESSAGES`.

STANDARD OUTPUT (STDOUT)

The standard output is equivalent to the input files with tab characters converted into the appropriate number of blanks.

EXIT STATUS

0 Successful completion.

>0 An error occurred.

SEE ALSO

tabs(1), unexpand(1).

User's Reference Manual

export(1)

NAME

`export` - export shell variables

SYNOPSIS

`export` [*name* ...]

DESCRIPTION

The shell built-in `export` marks the specified shell variable for export. This means that the name of this variable and its value will subsequently be known and accessible to all commands.

If `export` is invoked without arguments, the names of all shell variables that have been exported in the current shell are listed on standard output.

Exported variables are deleted when the shell in which they were defined and exported is terminated. Frequently used variables should therefore be defined and exported in the `$HOME/.profile` file.

Positional parameters and shell functions cannot be exported. Some standard variables of the shell are available in all subshells without needing to be exported. These include `HOME`, `IFS`, `PATH`, `PS1` and `PS2`. If you wish to assign some other value to these variables, and if the modified value is to be valid in every subshell, you will need to export these variables. Otherwise, the default value is valid in each subshell.

The shell built-in `set` displays the names and values of all variables defined in the current shell, including those that have not been exported.

The `env` command lists the names and values of all shell variables that are passed to each invoked command and each subshell. Variables such as `IFS`, `MAILCHECK`, `PATH`, `PS1` and `PS2` are contained in this output only if they have been exported with `export`.

OPERANDS

The Korn shell `ksh` supports additional options, and the values to be exported can be assigned a value [see `ksh(1)`].

name Name of the shell variable you want to export. This variable can also be assigned a value either before or after the `export` call.

You can specify as many shell variables as you like; they must be separated from one another by a blank.

name not specified:

All shell variables exported in the current shell are written to standard output. The output is generated in the following format:

`export` *name*

and for the Korn shell `ksh`:

name=*value*

or, if the `-p` option is set:

`export` *name*=*value*

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

The shell script `editor` is to display the value of the `EDITOR` variable. The script file contains the following:

```
: Invocation with sh editor
echo EDITOR: $EDITOR
```

The following interactive session demonstrates why the shell variable `EDITOR` has to be exported:

```
$ export
export HOME
export PATH
export PS1
$ EDITOR=vi
$ sh editor
EDITOR:
$ export EDITOR
$ export
export HOME
export PATH
export PS1
export EDITOR
$ sh editor
EDITOR: vi
```

When you define a shell variable, it is only known in the current shell. Since a shell script is always run by a subshell, variables for shell scripts have to be exported.

NOTES

Some differences in behavior may occur when using `export`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`env(1)`, `ksh(1)`, `set(1)`, `sh(1)`.

User's Reference Manual

expr(1)

NAME

`expr` - evaluate expressions

SYNOPSIS

`expr` *expression ...*

DESCRIPTION

`expr` interprets command line arguments as expressions and evaluates them in succession. The result of the evaluation is displayed on standard output.

You can use `expr` to compare strings with one another, for example, or to perform calculations with integers in the range of –9223372036854775808 to 9223372036854775807.

ARGUMENTS

expression

expression may either consist of one operand only, or two operands linked by an operator. Any arbitrary string can be specified as an operand. Strings that consist of digits only (0 to 9) are interpreted as integers. Integers preceded by a minus sign are interpreted as negative numbers.

The operands and operators are separated from one another by blanks or tabs. If the string you wish to use as an operand itself contains blanks or tabs, you must protect them from being interpreted as separators by enclosing either the entire string or just the blanks or tabs in single or double quotes ('...'or "..."). Shell metacharacters must also be quoted ('...'or "...") or individually escaped with a backslash [see `specialchar(5)`].

If *expression* consists of only one operand, the result of the evaluation will be the operand itself.

The following section describes how two operands can be linked. The operators are arranged in the order of increasing precedence; operators with equal precedence are enclosed in braces {...}.

A result of 0 stands for the value 0, not a null string.

op1 | *op2*

If the evaluation of *op1* returns neither a null string nor 0, the expression evaluates to *op1*; otherwise, *op2*.

op1 & *op2*

If the evaluation of neither *op1* nor *op2* returns a null string or 0, the result returned is *op1*; otherwise, 0.

op1 rel op2

rel may be one of the following relational operators:

<	less than
<=	less than or equal to
=	equal to
>=	greater than or equal to
>	greater than
!=	not equal to

If the condition is fulfilled, the comparison returns a result of 1. If the condition is not satisfied, the result of the comparison is 0. If both *op1* and *op2* are integers, the comparison is numeric; otherwise, they are interpreted as strings and compared alphanumerically.

op1 {+ -} *op2*

If *op1* and *op2* are both integers, the result returned is the sum of, or difference between, the two numbers. If either of the arguments is not an integer, `expr` issues an error message (see **ERROR MESSAGES**)

op1 {* / %} *op2*
When *op1* and *op2* are integers, the result is equal to the value obtained from the specified arithmetic operation:

`*` Multiplication

`/` Division (integers only)

`%` Remaindering

If either of the arguments is not an integer, `expr` issues an error message (see **ERROR MESSAGES**).

op1 `:` *op2*
The strings *op1* and *op2* are compared with one another, starting with the first character in each string and ending with the last character in *op2*. *op2* may be specified in the form of a simple regular expression [see `expressions(5)`]. If *op1* and *op2* match each other (i.e. from the first character in both strings to the last character in *op2*), `expr` usually returns the number of matching characters. However, if you enter the pattern \(...\) for *op2*, the part of *op1* that matches this pattern will be displayed (see *Example 7* and *Example 8*).

`match` *op1 op2*
Same as *op1* `:` *op2*.

`(...)` The use of parentheses `(...)` enables you to influence the sequence of evaluation. Parenthesized expressions are evaluated first even if they contain operators with lower precedence.

EXIT STATUS

`0` The expression is not incorrect and the evaluation returns neither 0 nor a null string.

`1` The evaluation of the expression returns either 0 or a null string.

`2` for invalid expressions or division by 0

`>2` Error

ERROR MESSAGES

`expr: non-numeric argument`
You are only allowed to specify integers as operands when using the arithmetic operators `+`, `-`, `*`, `/`, `%`.

`expr: division by zero`
You have tried to divide by 0.

`expr: syntax error`
You have made a syntax error when calling `expr`, e.g. you have not delimited operands and operators by blanks or tab.

`expr: RE error`
When using the relational operator : in an expression, you have not specified the second operand as a simple regular expression in the correct format.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In regular bracketed expressions, the `LC_COLLATE` environment variable governs the scope of character ranges, equivalence classes and collating elements, while the `LC_CTYPE` environment variable governs the scope of character classes. `LC_COLLATE` also governs the behavior of relational operators in string comparisons. Thus if the letters `a` and `ä` form part of an equivalence class, the expression

```
expr $var:[[=a=]]
```

returns a value of 1 if the value of the variable is `a` or `ä`.

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Simple calculation:

```
$ expr 21 + 9 '*'2 / 6
24
```

Example 2

A value of 1 is added to variable a in the following example:

```
$ echo $a
3
$ a=`expr $a + 1`
$ echo $a
4
```

Example 3

This example compares two environment variables:

```
$ echo $op1 $op2
text1 text2
$ expr $op1 = $op2
0
```

If the value of a variable is itself an operator used by expr, the comparison will not work. This problem can be solved by combining the variable with a "safe" character:

```
$ echo $op1 $op2
= =
$ expr $op1 = $op2
expr: syntax error
$ expr X$op1 = X$op2
1
```

Example 4

Displaying the number of characters in VAR:

```
$ echo $VAR
Hello
$ expr $VAR : '.*'
5
```

Example 5

Comparing two strings:

```
$ expr boycott : boy
3
```

Example 6

Comparison between two strings with the pattern specified as a regular expression. The caret ^ is a metacharacter for the shell and must therefore be escaped by a backslash \.

```
$ expr abc : [\^d-f]
1
```

Example 7

The path name of a file, e.g. `/usr/latino/parnassum/infinitum` is stored in variable `a`. To obtain the basic file name, i.e. `infinitum`, you enter:

```
$ expr $a : '.*/\(.*\)'□
infinitum
```

Example 8

If variable `a` contains either the path name or the basename of a file, you can extract the file basename as follows:

```
$ expr $a : '.*/\(.*\)'\| $a
```

NOTES

Many of the operators are also shell control operators, so they have to be escaped on the command line.

Each part of the expression is composed of separate arguments, so liberal usage of blank characters is required.

Examples:

Invalid	Valid
`expr 1+2`	`expr 1 + 2`
`expr "1 + 2"`	`expr 1 + 2`
`expr 1 + (2 * 3)`	`expr 1 + \( 2 \* 3 \)`

SEE ALSO

`ed`(1), `sh`(1).

User's Reference Manual

face(1)

NAME

`face -` executable for the Framed Access Command EnvironmentInterface

SYNOPSIS

`/home/vmsys/bin/face` [`–i` *init_file*] [`–c` *command_file*] [`–a` *alias_file*] [*file ...*]

DESCRIPTION

file is the full pathname of the file describing the object to be opened initially, and must follow the naming convention `Menu.`*xxx* for a menu, `Form.`*xxx* for a form, and `Text.`*xxx* for a text file, where *xxx* is any string that conforms to the Reliant UNIX system file naming conventions. The FMLI descriptor `lifetime` will be ignored for all frames opened by argument to `face`. These frames have a lifetime of `immortal` by default. If *file* is not specified on the command line, the FACE Menu will be opened along with those objects specified by the `LOGINWIN` environment variables. These variables are found in the user's `.environ` file.

DIAGNOSTICS

The `face` command will exit with a non-zero exit code if the user is not properly set up as a FACE user.

FILES

`$HOME/pref/.environ`

SEE ALSO

`env`(1), `environ`(5).

User's Reference Manual

factor(1)

NAME

`factor -` resolve number into prime factors

SYNOPSIS

Format 1

`factor`

Format 2

`factor` *number*

DESCRIPTION

The `factor` command resolves a number into its prime factors. `factor` can be invoked with or without an argument.

- Without an argument (**Format 1**):

  You can enter numbers to be factored interactively. `factor` then factors the number you enter and outputs its prime factors.

- With an argument (**Format 2**):

  `factor` resolves the argument into its prime factors and outputs them.

Format 1: Interactive factoring

`factor`

You enter numbers interactively. `factor` then resolves each number into its prime factors and outputs each prime factor as many times as it is contained in the number.

If you enter 0 or a non-numeric character, `factor` terminates.

Format 2: Factoring command line arguments

`factor` *number*

number

number must be a positive decimal integer ≤ 10^14. `factor` resolves the *number* argument into its prime factors and displays the *number* followed by each prime factor the appropriate number of times. *number* may also be specified in scientific notation, e.g. 1420 as 1.42e3 or as 1.42E3. If you enter 0 or a non-numeric character as *number*, the output is 0.

ERROR MESSAGES

`Ouch!`

The interactively entered number or the command line argument is not in the permissible range for `factor`. You may only specify positive numbers that are less than or equal to 10^14.

`Not an integer!`

The interactively entered number or command line argument is not an integer.

EXAMPLES

Example 1

Interactive factoring:

```
$ factor
123
     3
     41
77777777777777
```

```
Ouch!
1234
     2
     617
0
$
```

Example 2

Non-interactive factoring:

```
$ factor 100
100
     2
     2
     5
     5
```

User's Reference Manual

false(1)

NAME

`false -` return non-zero exit status

SYNOPSIS

```
false
```

DESCRIPTION

The `false` command simply returns a non-zero exit status and does nothing else. You can use `false` in shell scripts to generate the condition `false`.

You can use the companion command `true` to generate the condition `true` (exit status 0).

EXIT STATUS

Always non-zero

EXAMPLES

Example 1

The exit status of `false`:

```
$ false
$ echo $?
1
```

Example 2

The following shell script initiates an endless loop which you can terminate by pressing DEL.

```
until false
do
.
.
.
done
```

SEE ALSO

`sh`(1), `true`(1).

User's Reference Manual

fgrep(1)

NAME

`fgrep -` search for strings (fixed grep)

SYNOPSIS

`fgrep` [*option ...*] [*patternlist*] [*file ...*]

DESCRIPTION

`fgrep` reads lines from one or more files or from standard input and searches the lines for strings. Unless told otherwise (by options), `fgrep` copies every line containing one of the specified strings to standard output.

If you specify more than one input file, the relevant file name will be displayed before each output line.

The character strings for which `fgrep` is to search are specified either using *patternlist* or via the options `–e` *patternlist* or `–f` *patternfile*. One of these three arguments must be specified, multiple specifications are not allowed.

OPTIONS

The options `–c` and `–l` are mutually exclusive.

No option specified
: `fgrep` outputs all lines that match at least one of the strings specified in *patternlist*. If you specify more than one input file, each output line will be preceded by the name of the file in which the line was found.

`–b`
: (block) Each output line is preceded by the number of the block in which it was found.

  Each file is made up of 512-byte blocks which are numbered consecutively from 0.

  Option `–b` is sometimes useful in locating disk block numbers by context (see the *offset* argument for the `od` command, for example).

`–c`
: (count) `fgrep` outputs only the number of lines found (i.e. the lines that `fgrep` would have displayed without the `–c` option, see *Example 4*); the lines themselves are not displayed.

`–e` *list*
: (expression) Synonymous with *patternlist*. The `–e` option must always be specified if the first expression in *list* begins with a – (dash). The `–e` option ensures that such strings are not interpreted as an option but as a list of strings to be matched with the input lines.

`–f` *listfile*
: (file) `fgrep` reads the search strings from the named *listfile*. Each line in *listfile* is interpreted as a string.

`–h`
: (hidden) When searching multiple files, `fgrep` does not write the file name before each output line.

`–i`
: (ignore) `fgrep` does not distinguish between uppercase and lowercase.

`–l`
: (list) `fgrep` simply outputs the names of files that contain at least one of the matching lines. (These are the lines that `fgrep` would output if the `–l` option were omitted, see *Example 5*.) Each file name is printed just once. The lines themselves are not displayed.

`–n`
: (number lines) Each output line is preceded by its line number in the relevant input file. Line numbering starts at 1. If `fgrep` is reading from standard input, the line number refers to the standard input.

`–r`
: (recursive) Names that refer to directories are processed recursively; in other words, all the files and subdirectories in that directory are scanned as well.

`–v`
: (vice versa) `fgrep` outputs all lines that *do not* match any of the specified strings.

  In conjunction with option `–c`: `fgrep` prints only the number of lines that do not match.

  In conjunction with option `–l`: `fgrep` only outputs the names of files containing such lines.

`-x` (exact) `fgrep` outputs lines consisting solely of one of the specified strings.

In conjunction with option `-c`: `fgrep` only outputs the number of such lines.

In conjunction with option `-l`: `fgrep` outputs the names of files containing such lines.

`-y` (y – ignore) `fgrep` makes no distinction between uppercase and lowercase letters.

`--` End of the list of options. Must be specified if the first argument begins with `-`.

patternlist

List of strings that `fgrep` is to use when comparing input lines. The individual strings must be separated by newline characters. A newline character in *patternlist* functions like a logical OR operator.

If *patternlist* includes newline characters or other characters that have a special meaning for the shell, you must enclose *patternlist* in single quotes: '*patternlist*'.

file Name of the file that `fgrep` is to scan. You may name any number of files.

file not specified: `fgrep` reads input lines from the standard input.

grep, fgrep, and egrep

The `grep`, `fgrep` and `egrep` commands perform similar functions and are largely identical in terms of usage. The following section lists the most important differences between these commands.

`grep` processes simple regular expressions if the options `-E` or `-F` are not specified.

`fgrep` only processes character strings. `fgrep` can search for a large number of characters strings very efficiently: `fgrep` searches each individual line for all of the character strings. `fgrep` behaves in a similar way to `grep -F`.

`egrep` processes extended regular expressions. These include simple regular expressions, with the following exception: The simple regular expressions \(*regexpression*\) and \{...\} have no special meaning for extended regular expressions and are therefore not processed by `egrep`. `egrep` behaves in a similar way to `grep -E`.

EXIT STATUS

`0` Matching lines found

`1` No matching lines found

`>1` Syntax error or "Cannot open file". This exit status remains valid even if lines have been found in other input files.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_COLLATE` environment variable governs the collating sequence.

`LC_TYPE` governs character classes and character conversion (shifting, option `-i/-y`).

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The files `customer1` and `customer2` are the basis for all the examples below. Their contents are as follows:

`customer1`:

```
080685    999.98  20 Units  Item  038   Johnson Ltd.
120387   1240.25   3 Units  Item  023   Skinner Ltd.
180588    330.87   1 Units  Item  332   Skinner Ltd.
```

`customer2`:

```
morrow lance, 86 sherwood street, london w1
skinner robert, 16 garden hill, london ec3
```

Example 1

Output lines that match a string (without an option and with option `-i`):

```
$ fgrep Skinner customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:180588    330.87   1 Units  Item  332   Skinner Ltd.
```

If you also wish to display lines containing the word `skinner` in lowercase you enter:

```
$ fgrep -i skinner customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:180588    330.87   1 Units  Item  332   Skinner Ltd.
customer2:skinner robert, 16 garden hill, london ec3
```

Example 2

Search several strings (without an option and with the `-f` option):

```
$ fgrep 'Skinner
> Johnson' customer1 customer2
customer1:080685    999.98  20 Units  Item  038   Johnson Ltd.
customer1:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:180588    330.87   1 Units  Item  332   Skinner Ltd.
```

Alternatively, you could write both strings into a file called `names` (each string in a separate line) and then call `fgrep` as follows:

```
$ fgrep -f names customer1 customer2
```

Example 3

Output lines that do not contain the given string (option `-v`):

```
$ fgrep -v Skinner customer1 customer2
customer1:080685    999.98  20 Units  Item  038   Johnson Ltd.
customer2:morrow lance, 86 sherwood street, london w
customer2:skinner robert, 16 garden hill, london ec3
```

Example 4

Display the number of lines found (option `-c`):

To begin with, the number of lines containing the string `Skinner` are to be output for each input file.

```
$ fgrep -c Skinner customer1 customer2
customer1:2
customer2:0
```

Next the number of lines that do not contain the string `Skinner` is to be displayed.

```
$ fgrep -c -v Skinner customer1 customer2
customer1:1
customer2:2
```

Example 5

Display file names only (option `-l`):

First the names of files containing the string `Skinner` are to be output.

.fi $ fgrep -l Skinner customer1 customer2 customer1

.fi

Now the names of files with lines not containing the string `Skinner` are to be displayed.

```
$ fgrep -l -v Skinner customer1 customer2
customer1
customer2
```

Example 6

Display found lines with their line numbers (option `-n`):

```
$ fgrep -n -i skinner customer1 customer2
customer1:2:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:3:180588    330.87   1 Units  Item  332   Skinner Ltd.
customer2:2:skinner robert, 16 garden hill, london ec3
```

SEE ALSO

ed(1), grep(1), egrep(1), sed(1), sh(1), expressions(5).

User's Reference Manual

file(1)

NAME

`file -` check type of file

SYNOPSIS

Format 1

`file` [`–h`] [`–m` *magicfile*] [`–f` *ffile*] [*argfile ...*]

Format 2

`file –c` [`–m` *magicfile*]

DESCRIPTION

The `file` command takes a list of one or more files and classifies each file on the basis of its contents. The types distinguished by `file` include directories, special files, FIFO files, archive libraries, C source programs, executable programs, shell scripts, and normal text files.

Caution:

`file` looks up the magic number in the magic file to identify the target file. If it fails to identify the file by this method, it tries various plausibility checks. Consequently the results are not always reliable.

OPTIONS

Format 1: Classify a file

`file` [`–h`] [`–m` *magicfile*] [`–f` *ffile*] [*argfile ...*]

`–h` (hidden) If *argfile* is a symbolic link, the link will not be followed and the following error message will be printed:

file1`: symbolic link to` *file2*

`–m` *magicfile*
(magic) Causes `file` to use *magicfile* instead of the system file `/etc/magic` to identify the magic numbers of the files being classified.

`–f` *ffile*
`file` interprets the *ffile* argument as the name of a file which contains the names of all files to be examined. If this option is omitted, you must name at least one file to be classified.

argfile
Name of the file to be classified by the `file` command. If *argfile* is a symbolic link, `file` will follow the link and test the original file referenced by the symbolic link. You can name any number of files. If the `–f` option is omitted, you must name at least one.

Output

`file` writes its file classification to standard output. The following table lists the most important file types that `file` classifies:

Output	Meaning
`ASCII cpio archive`	Archive generated by `cpio`, option `–c`
`ascii text`	Text file
`assembler program text`	Assembler source program
`block special`	Block special file
`c program text`	C source program
`character special`	Character special file

commands text	Shell script
compressed data	Compressed file [see compress(1)]
cpio archive	Archive generated by cpio
current ar archive	Archive library [see ar(1)]
data	Data file
directory	Directory
ELF 32-bit	ELF object file; possible types:
core file	Core dump (file)
dynamic lib	Dynamic library
executable	Executable file
relocatable	Relocatable file
unknown	Type unknown
empty	Empty file
English text	Text file with English text
fifo	FIFO file
fortran program text	FORTRAN source program
[nt]roff, tbl, or eqn input text	File in nroff, troff, tbl or eqn format input text
packed data	compressed file [see pack(1)]
sccs	SCCS file
tar archive	Library, generated with tar
troff output	troff output file

Classification of executable programs goes into further detail. For example, if the symbol table has not been removed, file displays the message: not stripped [see strip(1)].

Mode of operation

file performs a series of tests on each input file and attempts to classify it on the basis of its contents. If it appears to be a text file, file examines an initial segment (the first 512 bytes) and tries to guess the language it was created in. The accuracy of the guess cannot be guaranteed, however.

If the input file is an executable program, it is identified as such, and further information is provided with respect to its contents. To do this, file searches the file for "magic numbers", i.e. for numeric constants or string constants that give an indication of the type of file. The file /etc/magic contains an explanation of these magic numbers.

Format 2: Check the magic file

file –c [–m *magicfile*]

–c (check) The magic file, by default the system file /etc/magic, is checked for format errors. However, if the –m option is specified, the magic file *magicfile* is checked instead.

–m *magicfile*
(magic file) Causes file to check the alternate magic file *magicfile* for format errors.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If

LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file list contains the following file names:

```
dir
letter
lib.a
prog.c
prog.s
prog.o
prog
```

The following command line will yield information on the contents of each classified file type:

```
$ file -f list
dir:            directory
letter:         ascii text
lib.a:          current ar archive
prog.c:         c program text
prog.s:         assembler program text
prog.o:         ELF 32-bit LSB relocatable 80386 Version 1
prog:           ELF 32-bit LSB executable 80386 Version 1
```

Thus dir is a directory, and letter contains normal ASCII text. lib.a is an archive library; prog.c contains a C program. prog.s probably contains an Assembler source program (no guarantee!). prog.o is an object module, prog an executable program, both in ELF format (Executable and Linking Format).

The same output could also have been obtained with a command line reading:

```
$ file dir letter lib.a prog.c prog.s prog.o prog
```

FILES

/etc/magic
: File containing a key to the magic numbers

SEE ALSO

cc(1), ls(1), strip(1), filehdr(4).

User's Reference Manual

find(1)

NAME

`find -` scan directories for files

SYNOPSIS

`find` *directory ... expression*

DESCRIPTION

`find` scans directories and all their subdirectories for files which match a specified search criterion. Files located in this way can be listed and/or processed with a command.

`find` locates not only ordinary files, but also directories, special files and FIFOs.

OPTIONS

directory

Name of the directory that is to be scanned. `find` will search through this directory and recursively through its subdirectories, provided you have read and execute permission (r and x bit) for them. More than one directory may be named.

expression

expression may be given as one or more primary expressions (primaries) of the kind listed below or as a combination of these primaries.

`find` applies *expression* to each file in the specified directories or subdirectories by evaluating primaries from left to right. Each primary is considered to have a "true" or "false" value for a file. The truth value of each expression and the next relational operator determine whether or not `find` will process the next primary.

Most primaries define conditions, i.e. they return a value of true if the current file satisfies the condition. The expressions:

`–print`

`–exec` *command* `\;`

`–ok` *command* `\;`

`–cpio` *device*

are not conditions. They cause actions to be executed, e.g. output of the file name or deletion of the file. When `find` processes one of these expressions, the associated action is always performed, regardless of whether the expression returns a value of true or false. If none of the above action primaries are specified, `–print` is executed by default.

Shell metacharacters that appear in *expression* must be escaped to ensure that they are passed to `find` and not interpreted by the shell.

The following section describes primaries; the method of combining primaries is shown thereafter.

Primaries as conditions

If you use options to find files with specific attributes (e.g. the files modified within a given period), the relevant option must come before the `–print` option (see below), as otherwise all files will be listed.

In the following discussion, *n* represents a decimal integer; +*n* means "more than *n*", –*n* means "less than *n*", and *n* means "exactly *n*".

`–name` *pattern*

True if *pattern* matches the name of the current file.

You can also use the usual shell metanotation for file name generation, but then *pattern* must be enclosed in single quotes, e.g. `find /usr/adam –name 'group.*'` [see `specialchar(5)`]. Only

file basenames may be specified; pathnames such as `text/tmp` are not permitted.

`-perm` *mode*
: True if the permissions for the current file are those described by *mode* (see `chmod`, Symbolic form). To compare the permissions, `find` generates a template without file permission bits and modifies it in accordance with the specifications for *mode*. If *mode* is prefixed by a minus sign, only the bits described in *mode* are compared with the permissions, and the expression evaluates true if they match.

In order to avoid incorrect interpretations, *mode* itself must not begin with -, i.e. you must always specify explicitly who you want to withdraw permissions from.

`-perm` *onum*
: True if the file permission flags exactly match the octal number *onum* (see `chmod`). 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` *character*
: True if the current file is of type *character*, where *character* can be:

`f` ordinary file

`d` directory

`b` block special file

`c` character special file

`l` symbolic links

`p` named pipe or FIFO file

`-fstype` *name*
: True if the file system to which the current file belongs is of type *name*, where *name* can be any of the following:

`hs`, `nfs`, `ufs` or `vxfs` (and also some special file systems such as `proc` or `fdfs`).

`-links` [±]*n*
: True if there are (more/less than) *n* links to the current file.

`-inum` [±]*n*
: True if the current file has an inode number greater than (+), less than (-), or equal to *n*. The expression `-inum +`*n* `-inum -`*m* can be used to find all inode numbers between *n* and *m*.

`-user` *login_name*
: True if the current file is owned by the user *login_name*.

`-nouser`
: True if the current file belongs to a user whose login name does not appear in the `/etc/passwd` file.

`-group` *group_name*
: True if the current file belongs to the group *group_name*.

`-nogroup`
: True if the current file belongs to a group that is not listed in the `/etc/group` file.

`-size` [±]*n*[`c`]
: `c` not specified:

True if the current file is (more/less than) *n* blocks long, where one block is 512 bytes.

`c` specified:

True if the current file is (more/less than) *n* bytes long.

`-atime` [±]*n*
: (access time) True if the current file was last accessed (more/less than) *n* * 24 hours ago. The access time of directories searched by `find` is changed by `find` itself.

`-mtime` [±]*n*
: (modification time) True if the current file was last modified (more/less than) *n* * 24 hours ago.

`-ctime` [±]*n*
: True if the inode of the current file was last modified (more/less than) *n* * 24 hours ago.

`-newer` *file*
: True if the current file is "newer" than the specified *file*, i.e. if it was created or modified at a later point in time.

`-local`
: True if the current file physically resides on the local system.

`-prune`
: Always yields true. Do not examine any directories or files in the directory structure below the *pattern* just matched. See `-name` *pattern* and *Example 5* and *6*, below.

  If `-depth` is specified, `-prune` will have no effect.

Action primaries

`-print`
: Always true. If this expression is present, it causes `find` to print the path name of the current file on standard output.

`-exec` *command* `\;`
: *command* is executed as soon as this expression is processed.

  You must terminate *command* with a blank and an escaped semicolon (\;). A set of braces {} can be used to represent the name of the current file.

  The expression `-exec` *command* `\;` is true if the executed *command* returns an exit status of 0. The truth value is significant when this expression is combined with other expressions.

`-ok` *command* `\;`
: Like `-exec`, except that `find` asks if the command is to be executed each time the expression is processed. The command is only executed if you respond with the character or string signifying "yes" in the current locale (see environment variable `LANG`).

`-cpio` *device*
: Always true. On processing this expression, `find` prints the contents of the specified file in `cpio` format on the specified *device* (see `cpio`).

  This option is obsolete and will not be supported in future releases.

Expressions that affect the behavior of find

`-depth`
: Always true. If present, this expression causes `find` to act upon entries in a directory before the directory itself. This can be useful if `find` is used with the `cpio` command to transfer files contained in directories for which you have no write permission.

`-follow`
: Always true. This expression causes symbolic links to be followed. It should not be combined with the `-type l` expression.

  Note that when you use `find` in conjunction with `cpio`, if you use the `-L` option with `cpio` you must use the `-follow` expression with `find`, and vice versa. Otherwise, there will be undesirable results.

`-mount`
: Always true. If this expression is present, it causes `find` to restrict its search to the file system containing the directory specified.

`-nonfs`
: If this option is set, `find` does not go into directories that have been mounted with NFS. If `find` is started with this option in an NFS directory, it ends without performing any action.

`-xdev`
: Like `-mount`.

Combining expressions

The expressions described above can be combined with one another as shown below. Please note the blanks used before and after the operators!

The operators are listed in order of decreasing precedence.

`\(` *expression...* `\)`
: Parentheses group expressions together. The parentheses themselves must be escaped with a backslash, since they have a special meaning for the shell.

`!` *expression*
: Negation, i.e. true if the expression evaluates to false.

expression expression or *expression* `-a` *expression*
: Logical AND, i.e. true if both expressions are true. If the first expression is false, `find` does not process the second expression.

  Example:

  If the second expression is

  `-exec` *command* `\;`

  the specified command will only be executed if the first expression is true.

expression `-o` *expression*
: Logical OR, i.e. true when either of the expressions is true. If the first expression is true, `find` does not process the second expression.

  Example:

  If the second expression is

  `-exec` *command* `\;`

  the command will only be executed if the first expression is false.

DIAGNOSTICS

Error messages without command termination (non-zero exit status)

`find: Cannot read dir` *dir*`: Permission denied`

Since you have no read permission for the *dir* directory, this directory cannot be scanned. `find` does not terminate after this error message, but continues scanning directories for which you do have the required access permissions (r bit).

`find: Cannot access` *dir*`/`*file*`: Permission denied`

Since you have no execute permission (x bit) for *dir*, this directory cannot be scanned. `find` does not terminate after this message but searches through those directories for which you do have the appropriate permissions (r and x bits).

Error messages without command termination (exit status 0)

`find: error while searching through/below` *dir*`: No such file or directory`

There is no directory named *dir*.

Error messages with command termination (non-zero exit status)

`find: missing conjunction`

You have either combined primaries incorrectly in the `find` call or specified more than one argument in a primary.

`find: no action specified`

You have not specified an action (`print`, `exec`, `ok`, `cpio`).

`find: incomplete statement`

You have forgotten the escaped semicolon \; which is required to terminate a command in an `-exec` or `-ok`

expression.

`find: insufficient number of arguments`

You have specified too few arguments.

`find: path-list predicate-list`

You have failed to specify the pathname list for *directory*.

EXAMPLES

Example 1

A simple example to illustrate how `find` works:

Display the path names of all files that

- are located in the current directory or in a subdirectory and
- have the name `tmp`.

```
$ find . –name tmp –print
```

In this case, *expression* comprises two primaries that are linked by a logical AND operator. `find` first checks the `–name tmp` primary for each file. If this expression is true (i.e. the current file is named `tmp`), `find` will also process the expression `–print`, which outputs the path names. Otherwise, `find` does not process the `–print` expression and remains silent.

Please note the sequence. If you entered:

```
$ find . –print –name tmp
```

instead, `find` would process the `–print` expression for each file, i.e. would output the path names of all files.

Example 2

A more complex example:

Display the path names of all files that

- are located in the current directory or its subdirectories, and
- are named `tmp` or have names ending in `.xx`.

```
$ find . \( -name tmp -o -name '*.xx' \) -print
```

Please note the use of blanks between the arguments!

`find` begins by processing the parenthesized specifications for each file. These comprise two primaries linked by a logical OR operator.

`find` first tests the `–name tmp` primary. If this expression is true (i.e. if the current file is named `tmp`), `find` does not process the expression `–name '*.xx'` at all; this is because the result of the expression in parentheses will be true in any case. If `–name tmp` evaluates to false, `find` tests the second primary, i.e. `–name '*.xx'`. If this expression is true (i.e. the file name ends in `.xx`), the parenthesized expression will also evaluate to true; otherwise, the result of the expression in parentheses is false.

If the parenthesized expression is true, `find` then processes the `–print` expression, i.e. outputs the file names (AND operator; see *Example 1*). If false, `find` does not process the `–print` expression, i.e. remains silent.

Example 3

Search all entries in the `/usr/santaclaus` directory and its subdirectories and print the files not owned by `santaclaus`.

```
$ find /usr/santaclaus ! –user santaclaus –print
```

Example 4

Delete all files that

- are named `tmp` or have names ending in `.xx`, and
- have not been accessed for more than 7 days.

Confirmation is to be requested before the removal of each file.

```
$ find / \( -name tmp -o -name '*.xx' \) -atime +7 -ok rm {} \;
```

Example 5

Recursively print all file names in the current directory and below, but skipping all SCCS directories:

```
$ find . –name SCCS –prune –o –print
```

Example 6

Recursively print all file names in the current directory and below, skipping the contents of SCCS directories, but printing out the SCCS directory name:

```
find . –print –name SCCS –prune
```

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In file name generation patterns in square brackets (such as `find . –name [[=a=]]* –print`), the `LC_COLLATE` environment variable governs the scope of character ranges, equivalence classes and collating elements, and the `LC_TYPE` environment variable governs the scope of character classes.

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

FILES

`/etc/group`
: File containing group names

`/etc/passwd`
: File containing login names

SEE ALSO

`chmod`(1), `cpio`(1), `ln`(1), `sh`(1), `test`(1), `stat`(2), `fs`(4).

flchk(1)

NAME

`flchk -` check floppy disk label area (floppy check)

SYNOPSIS

`/usr/sbin/flchk [ –o` *owner* `] [ –v` *version* `] [ –i ]`

DESCRIPTION

`flchk` is used to check the label area of the floppy disk currently in the drive. This label area, which is in the first few tracks of the floppy disk, is compared with the arguments that you specify in the command line. The format of a floppy disk is discussed in the `flinit(1)` command description.

The following can be recorded in the label area:

- the owner of the floppy disk
- the volume serial number.

`flchk` resides in the directory `/usr/sbin`.

The `flinit(1)` command can be used to make an entry in the label area of a floppy disk; the `fldisp(1)` command displays this entry.

OPTIONS

`–o` *owner*

`flchk` compares the given *owner* with the corresponding entry on the floppy disk. If the owners are different, `flchk` asks if the difference should be ignored. If you answer "n", `flchk` issues the message:

```
incorrect floppy inserted
```

and terminates with exit status 1.

`–v` *version*

`flchk` compares the given volume serial number *version* with the entry on the floppy disk. If the numbers are different, `flchk` asks if the difference should be ignored. If you answer "n", `flchk` issues the message:

```
incorrect floppy inserted
```

and terminates with exit status 1.

`–i` Modifies how the command behaves with options `–v` and/or `–o` as follows:

If the specified volume serial number or owner does not match the corresponding entry on the floppy disk, `flchk` issues an error message and terminates immediately with exit status 1 without requesting confirmation.

Use of the flchk command

`flchk` is typically used by the menu system when processing floppy disks. The entry in the label area enables a program to identify a floppy disk. Furthermore, in the menu system, only the owner recorded in the label area can access a floppy disk. Data is thus protected against access by unauthorized users, provided the access attempt is made in the menu system.

EXIT STATUS

`0` The entries in the label area of the floppy disk in the drive match the command line arguments.

`1` One entry/both entries in the label area of the floppy disk in the drive is/are inconsistent with the command line arguments.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

Answers to yes/no queries must be given in the language appropriate to the current locale.

LC_CTYPE governs character classes and character conversion (shifting).

If LC_MESSAGES or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

You wish to check whether the correct floppy disk is in the drive. The correct owner is john; the volume serial number is 1.

```
$ flchk –v 1 –o john
requested: VSN  000001  owner john
existing:  VSN  000003  owner eric
Should the difference be ignored? (y/n) > n
flchk: Incorrect floppy inserted!
```

or

```
$ flchk –v 1 –o john –i
requested: VSN  000001  owner john
existing:  VSN  000003  owner eric
flchk: Incorrect floppy inserted!
```

SEE ALSO

fldisp(1), flinit(1).

User's Reference Manual

fldisp(1)

NAME

`fldisp -` display floppy disk label area (floppy display)

SYNOPSIS

`/usr/sbin/fldisp [–o] [–v]`

DESCRIPTION

`fldisp` displays the entries in the label area of the floppy disk currently in the drive. This area is in the first few tracks of the floppy disk. The format of a floppy disk is discussed in the `flinit(1)` command description.

The label area of a floppy disk may contain the following entries, which you can display with `fldisp`:

- the owner of the floppy disk,
- the volume serial number.

`fldisp` resides in the directory `/usr/sbin`.

Entries in the label area of a floppy disk are made with `flinit`.

OPTIONS

No option specified:
`fldisp` displays the name of the owner and the volume serial number.

`–o` Displays the name of the owner.

`–v` Displays the volume serial number.

Use of the fldisp command

`fldisp` is used by the menu system when processing floppy disks. The entry in the label area enables a program to identify a floppy disk. Furthermore, in the menu system, only the owner recorded in the label area can access a floppy disk. Data is thus protected against access by unauthorized users, provided the access attempt is made in the menu system.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

Answers to yes/no queries must be given in the language appropriate to the current locale.

`LC_CTYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

To display the label area of a floppy disk, you enter:

```
$ fldisp
VSN:   000001
owner: john
```

The owner of the floppy disk is `john`; the volume serial number is `1`.

Example 2

You now wish to check the owner of a floppy disk:

```
$ fldisp –o
```

`john`

The owner of this floppy disk is `john`.

SEE ALSO

`flchk`(1), `flinit`(1).

User's Reference Manual

flinit(1)

NAME

`flinit -` write floppy disk label area (floppy initialization)

SYNOPSIS

`/usr/sbin/flinit` [–o *owner*] [–v *version*]

DESCRIPTION

`flinit` initializes the label area of the floppy disk currently in drive 0. This area is at the beginning of the floppy disk (track 0).

The following can be recorded in the label area:

- the owner of the floppy disk,
- the volume serial number.

Entries in the label area of a floppy disk can be displayed and checked with the `fldisp(1)` and `flchk(1)` commands, respectively.

`flinit` resides in the directory `/usr/sbin`.

Before the call

The floppy disk must be in drive 0, and it must not be write-protected. `flinit` only supports floppy disks in drive 0.

If you are using a new floppy disk, you will have to format it first. This must be done with the command `flformat /dev/rdsk/f0...t`.

depending on how you wish to have the floppy disk formatted. Any of the available floppy disk formats [see `flformat(1M)`] may be used. Only then will the floppy disk be formatted in a manner that will allow you to access the label area with `flinit`.

OPTIONS

No option specified:

A null string is entered as the volume serial number *version*.

owner is entered as 14 blanks.

–o *owner*

Here you can specify an owner name up to 14 characters long.

–o *owner* not specified: *owner* is entered as 14 blanks.

–v *version*

version is an alphanumeric string (max. 6 characters) that is written to the label area. If the volume serial number comprises digits only, it is stored right-aligned, and padded to the left with leading zeros as required. If the specified volume serial number includes characters other than digits, it is stored left-aligned.

–v *version* not specified:

A null string is entered as the volume serial number.

The format of a floppy disk

Floppy disks that are formatted with `flformat` and initialized with `flinit` are divided into two areas:

- the label area, which you can access via the special file name `/dev/dsk/f0...t` or, for character-oriented (raw) access, `/dev/rdsk/f0...t`. The label area may contain information about the owner of the disk and about the volume serial number of the disk.

  Caution:

When you access a floppy disk via one of the special files listed above, you can read from and write to the entire floppy disk, as the device driver does not restrict access to just the label area. There is thus a potential danger of inadvertently overwriting data that follows the label area.

- the data area, which you can access via the file name `/dev/dsk/f0...` (no t at the end). This is the area in which your data is stored.

Use of the flinit command

`flinit` is used by the menu system when processing floppy disks. The entry in the label area enables a program to identify a floppy disk. Furthermore, in the menu system, only the owner recorded in the label area can access a floppy disk. Data is thus protected against access by unauthorized users, provided the access attempt is made in the menu system.

Caution:

Commands such as `cpio` and `tar` do not check the label area. In other words, the `flinit` command cannot be used as an all-purpose protection against unauthorized access.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

Answers to yes/no queries must be given in the language appropriate to the current locale.

`LC_CTYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

You can enter the user name `john` and the volume serial number `1` in the label area with:

```
$ flinit –v 1 –o john
```

and then use the `fldisp` command to check whether the entries have been made correctly.

The entry in the label area is shown as:

```
VSN:   000001
owner: john
```

FILES

`/dev/dsk/f0...t`
: Special file used to access the label area of a floppy disk.

`/dev/dsk/f0...`
: Special file used to access the data area of a floppy disk.

SEE ALSO

`flchk(1)`, `fldisp(1)`, `flformat(1M)`.

User's Reference Manual

fmli(1)

NAME

`fmli -` invoke FMLI (Form and Menu Language Interpreter)

SYNOPSIS

`fmli` [*option ...*] *file ...*

DESCRIPTION

The Form and Menu Language Interpreter FMLI provides a framework for developers to write applications and application interfaces that use menus and forms. The `fmli` command invokes (FMLI) and opens the frame(s) specified by the file argument. For a detailed description of FMLI you should refer to the *Programmer's Guide: Character User Interface* and the *Programmer's Guide: Character User Interface, Supplement*.

OPTIONS

No option specified:

`fmli` invokes the Form and Menu Language Interpreter and opens the frame or frames specified by the *file* argument.

`–a` *alias_file*

alias_file is the name of a file which contains synonyms and abbreviations that serve to simplify file references to objects or devices with lengthy pathnames. Each line in *alias_file* must be an alias definition of the form `alias=`*pathname*. The aliases defined in this file can be referenced with `$alias` in definition files, thus making it easier to reference objects or devices with long pathnames. In addition, they can be used to define a search path (similar to `$PATH` in the Reliant UNIX system shell).

`–c` *command_file*

command_file is the name of a file in which default FMLI commands can be disabled, and new application-specific commands can be defined. The contents of this file are reflected in the FMLI Command Menu.

`–i` *initialization_file*

initialization_file is the name of a file in which one or more of the following characteristics of the application as a whole can be specified:

- atransient introductory frame displaying product information
- abanner, its position, and other elements of the banner line
- color attributes for all elements of the screen
- function keys known as Screen Labeled Keys (SLKs) and their layout on the screen.

file *file* must begin with one of the three recognized object names, `Menu.`, `Form.` or `Text.`, and the file must also have a defined data format (see *Programmer's Guide: Character User Interface*).

ERROR MESSAGES

`Initial object must be specified`

No frame file *file* was supplied as an argument to the `fmli` command.

`Can't open object file`

`fmli` returns the above error message and exits if the specified *file* does not exist or is not readable,

`I do not recognize that kind of object`

The name of the specified *file* does not start with one of the three recognized object names (`Menu.`, `Form.`, or `Text.`), or the data contained in it is not in the proper format.

ENVIRONMENT VARIABLES

`LOADPFK`

On terminals with programmable function keys, `fmli` usually downloads its equivalent character sequences for using function keys into the terminal's programmable function key memory, wiping out any function key settings the user may already have defined. Setting `LOADPFK=NO` in the environment will prevent this downloading.

`COLUMNS`
Can be used to specify an alternate number of columns to override the screen width defined for the terminal set in `TERM`.

Example:

On a terminal with a 132-column mode, invoking FMLI with the command line:

```
COLUMNS=132 fmli filename
```

allows a screen width of 132 columns to be used.

`LINES`
Can be used to specify an alternate number of lines to override the screen length defined for the terminal set in `TERM`.

EXAMPLES

Example 1

Invoking `fmli` with a file named according to the file name conventions:

```
fmli Menu.start
```

Example 2

Invoking `fmli` with the initialization file `init.app`:

```
fmli –i init.app Menu.start
```

SEE ALSO

Programmer's Guide: Character User Interface.

Programmer's Guide: Character User Interface, Supplement.

User's Reference Manual

fmt(1)

NAME

`fmt` - simple text formatter

SYNOPSIS

`fmt` [*option* ...] [*file*] ...

DESCRIPTION

`fmt` is a simple text formatter which can be used to format text files or text read in from the standard input. It is capable of splitting, joining, and filling lines to produce output lines of a specified length. The result is written to standard output.

OPTIONS

No option specified:

`fmt` joins and splits input lines to produce output lines of up to 72 characters and writes the new lines to standard output.

`-c` (crown margin mode) `fmt` preserves the indentation of the first two lines within a paragraph and aligns the left margin of each subsequent line within the paragraph with that of the second line. This option is useful for multiple tagged paragraphs.

`-s` (split lines only) `fmt` does not join short lines to form longer ones. This prevents sample lines of program code, and other such formatted text, from being inappropriately combined.

`-w` *width*

Output lines are to be up to *width* columns long. If this option is not specified, `fmt` produces output lines with a default width of 72 characters.

file Name of the text file to be reformatted. You may name more than one file; in this case the `fmt` options you specify are applied to all files, and all the files are concatenated.

file not specified:

`fmt` reads from standard input.

Mode of operation

Indentation is preserved in the output, and input lines with differing indentation are not joined (unless the `-c` option is used).

Blank lines and the spacing between words are preserved in the output.

Words longer than *width* characters are not split. The maximum permissible line length is exceeded in this case.

Lines beginning with a . (dot) are not filled by `fmt` (for compatibility with `nroff`).

Lines starting with "From:", e.g. header lines in mail (see `mail`), are also preserved.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

The file named `text` is displayed first in its original state with the `cat` command and then formatted to a width of 40 columns with `fmt`:

```
$ cat text
It is evening, and time for the flowers to close
their petals. Give me leave to sit
by your side and bid my lips do the work that can
be done in silence,
and in the dim light of the stars.
$ fmt –w 40 text
It is evening, and time for the flowers
to close their petals. Give me leave to
sit by your side and bid my lips do the
work that can be done in silence, and in
the dim light of the stars.
```

SEE ALSO

vi(1), nroff(1M).

User's Reference Manual

fmtmsg(1)

NAME

`fmtmsg -` display formatted messages

SYNOPSIS

`fmtmsg` [*option ...*] *message_text*

DESCRIPTION

`fmtmsg` writes formatted messages to standard error (stderr) or to the system console.

A formatted message consists of up to five standard components. The classification and subclass components are not displayed as part of the standard message, but they define the cause and source of the message and direct the output of the formatted message.

OPTIONS

No option specified:

`fmtmsg` prints a list of valid options.

–`c` *class*

Describes the source of the message. *class* can be any of the following keywords:

`hard` The source of the condition is hardware.

`soft` The source of the condition is software.

`firm` The source of the condition is firmware.

–`u` *subclass*

subclass is a list of keywords that further define the message and direct the output of the message. The individual keywords must be delimited by commas.

Valid keywords for *subclass* are:

`appl` The condition originated in an application. Not to be used in combination with `util` or `opsys`.

`util` The condition originated in a utility. Not to be used in combination with `appl` or `opsys`.

`opsys`
The message originated in the kernel. Not to be used in combination with `appl` or `util`.

`recov`
The application will recover from the condition. Not to be used in combination with `nrecov`.

`nrecov`
The application will not recover from the condition. Not to be used in combination with `recov`.

`print`
Write the message to standard error.

`console`
Write the message to the system console.

`print` and `console` may also be specified together.

–`l` *label*

(label) Identifies the source of the message.

–`s` *severity*

Indicates the seriousness of the error. The keywords and definitions of the standard severity levels are:

`halt` The application has encountered a severe error and is halting.

`error`
The application has detected an error.

`warn` The application has detected a condition that is out of the ordinary and might be a problem.

`info` The application is providing information about a condition that is not an error.

`-t` *tag*
A string containing an identifier for the message.

`-a` *action*
A string describing the first step in the error recovery process. This string must be written so that the entire action argument is interpreted as a single argument. `fmtmsg` precedes each action string with the `TO FIX:` prefix.

message_text
The text of the message. This must be written so that the entire argument is interpreted as a single argument.

ENVIRONMENT VARIABLES

`MSGVERB`
`MSGVERB` is set by the system administrator in the `/etc/profile` for the system. Users can override the set value by assigning a new value to `MSGVERB` in their own `.profile` files.

`MSGVERB` tells `fmtmsg` which message components to select when writing to stderr. The value of `MSGVERB` is a colon-separated list of optional keywords. `MSGVERB` can be set as follows:

`MSGVERB=[`*keyword*`[:`*keyword*`[:...]]]`

`export MSGVERB`

The valid keywords are: `label`, `severity`, `text`, `action`, and `tag`. The keywords may appear in any order. If `MSGVERB` contains a keyword for a component and if the argument specified for the component in the `fmtmsg` call is not a null string, `fmtmsg` will include that component in the message when writing the message to stderr. If `MSGVERB` does not contain a keyword for a message component, that component will not appear in the displayed message.

If `MSGVERB` is not defined, if its value is the null string, if its value is not of the correct format, or if it contains illegal keywords, `fmtmsg` selects all components.

`MSGVERB` is only evaluated for messages that are to be written to stderr. All message components are included in console messages.

`SEV_LEVEL`
This variable can be used in shell scripts.

`SEV_LEVEL` defines additional severity levels for use by `fmtmsg`. The standard severity levels shown below cannot be modified. The predefined levels are:

`0` (no severity is used)

`1` HALT

`2` ERROR

`3` WARNING

`4` INFO

`SEV_LEVEL` is set as follows:

`SEV_LEVEL=[`*description*`[:`*description*`[:...]]]`

`export SEV_LEVEL`

description is a comma-separated list containing three fields:

description=*severity_keyword*,*level*,*print_string*

severity_keyword can be used with the `-s` option. *level* is a string that evaluates to a positive integer which is passed on to `fmtmsg` if *severity_keyword* is used. The values 0, 1, 2, 3 and 4 are reserved for the standard severity levels and cannot be used.

print_string is the character string printed by `fmtmsg` whenever the severity value *level* is used.

If SEV_LEVEL is not defined, or if its value is null, no severity levels other than the defaults are available. If the colon-separated list defining the variable contains syntax errors, the invalid part is ignored.

EXIT STATUS

0 All functions were executed successfully.

1 The command contains a syntax error, an invalid option, or an invalid argument to an option.

2 The function executed with partial success, but the message was not displayed on standard error.

4 The function executed with partial success, but the message was not displayed on the system console.

32 None of the requested functions were executed successfully.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The purpose of the following shell script is to make the command ls –l available under the abbreviated name ll. In this case, however, instead of using the standard error messages of the ls command, these messages are intercepted by the addition of 2>/dev/null. Errors that occur are detected from the return code $? of the command, and a user-defined error message is generated with fmtmsg when required.

```
$ cat ll
ls –l $* 2>/dev/null
case $? in
     0) echo "That was a listing of directory `pwd`/$*";;
     *) fmtmsg –u print –l "ll" –s error –t "(DIR may be invalid)" –a \
 "Try using another directory" "An error has occurred!"
esac
$ ll junkmail
ll: ERROR: An error has occurred!
   TO FIX: Try using another directory  (directory may be invalid)
$ ll junk
total 4
-rw-------   1 cmd      cmdag        690 Mar 18 14:00 file.Z
-rw-r--r--   1 cmd      cmdag         63 Mar 13 09:29 initvalues
That was a listing of directory /home/cmds/john/junk
```

SEE ALSO

addseverity(3C), fmtmsg(3C).

User's Reference Manual

fold(1)

NAME

`fold -` fold long lines

SYNOPSIS

`fold` [*option ...*] [*file ...*]

DESCRIPTION

`fold` reads the contents of text files or standard input and splits (folds) long lines so that they do not exceed the default or specified line width. The default line width is 80 characters per line. `fold` splits lines by inserting a new-line character.

The backspace, carriage-return and tab characters are treated specially in this case:

- Backspace decrements the character count by 1.
- Carriage-return resets the character count to 0.
- A tab character advances the character count to the next tab position.

OPTIONS

No option specified:
`fold` splits the input lines so that no line exceeds a maximum width of 80 characters (default).

`–b` (b – byte) `fold` counts bytes instead of characters. A line is never split within a character.

`–s` (s – space) `fold` splits the line after the last character within the maximum line length.

`–w` *width*
Defines the maximum width of a line, where *width* is the number of characters or bytes per line.

width should be a multiple of 8 if tabs are present in the input line.

`––` If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file Name of the text file to be folded. You may name several such files, in which case the specified options will be applied to each file.

file not specified: `fold` reads from standard input.

ERROR MESSAGES

`Bad number for fold`

This error message is displayed if you specify an invalid value for *width*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Printing a file named `notes` on standard output.

The following result is obtained with the `cat` command:

```
$ cat
Notes:
Remind Carl to send in a bid within the next two weeks. The new deadline
for the evaluation of all bids is the end of June.
```

Using the `fold` command, you have the option of redefining the length of the output lines.

The following command folds the contents of the `notes` file so that each line has a maximum width of 40 characters:

```
$ fold -s -w 40 notes
```

```
Notes:
Remind Carl to send in a bid within the
next two weeks. The new deadline for the
evaluation of all bids is the end of
June.
```

SEE ALSO

`cut`(1), `pr`(1).

User's Reference Manual

fsirand(1)
BSD Compatibility Package

NAME

`fsirand -` install random inode generation numbers

SYNOPSIS

`/usr/ucb/fsirand [ –p ]` *special*

DESCRIPTION

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

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

The `–p` option prints out the generation numbers for all the inodes, but does not change the generation numbers.

SEE ALSO

`fsck(1M)` .

gcore(1) - ksh(1)

User's Reference Manual

gcore(1)

NAME

`gcore -` get core images of running processes

SYNOPSIS

`gcore [ –o` *filename* `]` *process_id ...*

DESCRIPTION

`gcore` creates a core image of each specified process. Such an image may be used with debuggers such as `fsdb`.

OPTIONS

`–o` not specified:
: `gcore` creates a core image for each process whose process ID is specified in the argument list. The core image is stored in the file `core.`*process_id*.

`–o` *filename*
: (output file) The core image is stored in the file *filename*`.`*process_id* instead of `core.`*process_id*.

process_id
: Process ID of the process for which a core image is to be created. More than one process ID may be specified.

EXAMPLES

First, print an overview of all running processes with the `ps` command:

```
$ ps
   PID TTY      TIME COMD
 27623 tty004   0:02 sh
```

```
 27691 tty004  0:00 ps
```

The process ID of each process is shown in the first column. The `gcore` command can now be called to store a core image of the `sh` process in a file:

```
$ gcore 27623
gcore: core.27623 dumped
$ ls –l
total 64
-rw-r--r--   1 cmd      cmdog      30016 Mar 19 13:08 core.27623
-rw-------   1 cmd      cmdag        690 Mar 18 14:00 file
-rw-r--r--   1 cmd      cmdag         63 Mar 13 09:29 startup
```

As you can see, the core image is saved in the file `core.27623` and can be processed further as required.

FILES

`./core.`*process_id*

File containing the core image for the process identified by *process_id*.

SEE ALSO

`kill`(1), `ps`(1), `fsdb`(1M), `ptrace`(2).

User's Reference Manual

gencat(1)

NAME

`gencat` - generate a binary encoded message catalog

SYNOPSIS

`gencat` [*option ...*] *catfile msgfile ...*

DESCRIPTION

The `gencat` utility merges the contents of a message text source file into a binary encoded message catalog.

The message catalog is created if it does not already exist. If it does exist, its messages are included in the new message catalog. If the set numbers and message numbers of existing and newly defined message texts match, `gencat` replaces the old message texts currently contained in the message catalog with the new message texts defined in the message text file.

Message catalogs produced by `gencat` are binary encoded, which means that their portability cannot be guaranteed between different types of machine. Thus, just as C programs need to be recompiled for each type of machine, message catalogs must be recreated with `gencat`.

A message text source file can contain either set and message numbers or simply message numbers. In the later case, the set `NL_SETD` is assumed for the set numbers.

OPTIONS

The options `–m` and `–X` are mutually exclusive; the last option set takes precedence.

`–l` If *catfile* already exists, information on messages contained in it will be written to standard output in the following format:

With the `–X` option:

`SET` *set_num*, `MESSAGE` *msg_num*, `TEXT` *msg_txt* □
*

Without the `–X` option:

`SET` *set_num*, `MESSAGE` *msg_num*, `OFFSET` *offset*, `LENGTH` *msg_len message_text* □
*

offset indicates the distance from the beginning of the file to the start of the message text, i.e.

first message:

offset(1) = 0

*n*th message:

offset(*n*) = *offset*(*n*–1) + *msg_len*(*n*)

`–m` The `–m` option ensures compliance with the X/Open standard.

The `–m` option is supplied for compatibility with previous versions of `gencat` released in a number of specialized internationalization products. This option will cause `gencat` to build a single file *catfile* which is compatible with the format catalogs produced by the earlier versions. The retrieval routines detect the type of catalog they are using and act appropriately.

The `–m` option corresponds to the default behavior of `gencat` and need not be explicitly set.

`–X` `gencat` builds upon the `mkmsgs` utility. The `gencat` database comprises two files: *catfile*`.m`, which is an `mkmsgs` format catalog, and *catfile*, which contains the information required to translate a set and message number into a simple message number that can be used in a call to `gettxt`.

`gettxt` expects the catalogs to be located in a subdirectory under `/usr/lib/locale`. This restriction is lifted by placing a symbolic link to the catalog in the directory `/usr/lib/locale/Xopen/LC_MESSAGES` when the catalog is opened. This link enables `gettxt` to access the catalog. The link is removed when the catalog is closed.

–– End of the list of options. Must be specified if *catfile* begins with –.

No option specified:
`gencat` behaves as described under the –`m` option.

catfile
Name of the binary encoded message catalog that `gencat` is to generate from the message text file *msgfile*. If a dash (–) is specified for *catfile*, `gencat` prints to standard output.

msgfile
Name of the message text file that was generated by `extract(1)`. `gencat` generates a binary encoded message catalog from this file.

If a dash (–) is specified for *msgfile*, `gencat` reads from standard input. More than one message text file may be specified.

FORMAT OF A MESSAGE CATALOG

A message catalog created by `gencat` contains the following structures in the given order:

- The catalog header, `CAT_HDR`, consisting of:
  - the file's magic number
  - the number of sets in the message file
  - the space (in bytes) needed to load the file, excluding the length of the file header
  - the position at which the message headers begin, excluding the length of the file header
  - the position at which the message texts begin, excluding the length of the file header
- A set header, `CAT_SET_HDR`, for each existing set. This header consists of:
  - the set number
  - the number of messages in the set
  - the initial offset in the table
- A message header, `CAT_MSG_HDR`, for each existing message. This header consists of:
  - the message number
  - the length of the message in bytes
  - the message offset in the table
- The individual message texts, delimited by \0.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Generating a binary encoded message catalog `catfile` from the message text file `msgfile`, which was generated earlier with `extract(1)`.

```
$ gencat catfile msgfile
```

SEE ALSO

`dumpmsg(1)`, `extract(1)`, `iconv(1)`, `mkmsgs(1)`, `catclose(3C)`, `catgets(3C)`, `catopen(3C)`, `gettxt(3C)`, `nl_types(5)`.

Programmer's Guide: *Internationalization – Localization*.

User's Reference Manual

getconf(1)

NAME

`getconf` - get configuration values

SYNOPSIS

`getconf` *system_var*

`getconf` `-64s` *system_var*

`getconf` *path_var pathname*

DESCRIPTION

In the first synopsis form, `getconf` writes the value of the variable specified by the *system_var* operand to the standard output.

In the second synopsis form, `getconf` supplies 64-bit values of the variables `LONG_MAX` and `ULONG_MAX`.

In the third synopsis form, `getconf` writes the value of the variable specified by the *path_var* operand to the standard output. This value is valid within the path specified by the *pathname* operand.

The value of each configuration variable is output as if it were determined by calling the function from which it was defined (see **OPERANDS**). The value reflects the conditions in the current execution environment.

If the specified variable is valid, but is not defined on the system, `getconf` writes `undefined` to the standard output.

OPERANDS

path_var

Name of a configuration variable whose value can be obtained using the `pathconf(2)` function. All of the values listed in the following table are supported:

```
LINK_MAX
MAX_CANON
MAX_INPUT
NAME_MAX
PATH_MAX
PIPE_BUF
POSIX_CHOWN_RESTRICTED
POSIX_NO_TRUNC
POSIX_VDISABLE
```

Note: All variable names can be specified with or without a leading underscore (_).

pathname

Pathname for which the variable specified by *path_var* is to be determined.

system_var

Name of a configuration variable whose value can be obtained using the `confstr(3C)` or `sysconf(3C)` functions. All of the values listed in the following table are supported:

```
ARG_MAX
BC_BASE_MAX
BC_DIM_MAX
BC_SCALE_MAX
BC_STRING_MAX
BEILEN_MAX
CHARCLASS_NAME_MAX
CHAR_BIT
CHAR_MAX
```

```
CHAR_MIN
CHILD_MAX
CLK_TCK
COLL_WEIGHTS_MAX
CS_PATH
EXPR_NEST_MAX
INT_MAX
INT_MIN
LINE_MAX
LONG_BIT
LONG_MAX
LONG_MIN
NGROUPS_MAX
NL_ARGMAX
NL_LANGMAX
NL_MSGMAX
NL_NMAX
NL_SETMAX
NL_TEXTMAX
NZERO
OPEN_MAX
POSIX2_BC_BASE_MAX
POSIX2_BC_DIM_MAX
POSIX2_BC_SCALE_MAX
POSIX2_BC_STRING_MAX
POSIX2_CHAR_TERM
POSIX2_COLL_WEIGHTS_MAX
POSIX2_C_BIND
POSIX2_C_DEV
POSIX2_C_VERSION
POSIX2_EXPR_NEST_MAX
POSIX2_FORT_DEV
POSIX2_FORT_RUN
POSIX2_LINE_MAX
POSIX2_LOCALEDEF
POSIX2_RE_DUP_MAX
POSIX2_SW_DEV
POSIX2_UPE
POSIX2_VERSION
POSIX_ARG_MAX
POSIX_CHILD_MAX
POSIX_JOB_CONTROL
POSIX_LINK_MAX
POSIX_MAX_CANON
POSIX_MAX_INPUT
POSIX_NAME_MAX
POSIX_NGROUPS_MAX
POSIX_OPEN_MAX
POSIX_PATH_MAX
POSIX_PIPE_BUF
POSIX_SAVED_IDS
POSIX_SSIZE_MAX
POSIX_STREAM_MAX
POSIX_TZNAME_MAX
```

POSIX_VERSION
RE_DUP_MAX
SCHAR_MAX
SCHAR_MIN
SHRT_MAX
SHRT_MIN
SSIZE_MAX
STREAM_MAX
TMP_MAX
TZNAME_MAX
UCHAR_MAX
UINT_MAX
ULONG_MAX
USHRT_MAX
WORD_BIT
XOPEN_CRYPT
XOPEN_ENH_I18N
XOPEN_SHM
XOPEN_VERSION
XOPEN_XCU_VERSION
XOPEN_XPG2
XOPEN_XPG3
XOPEN_XPG4

The symbol PATH is also recognized. It returns the same value as the confstr(3C) value CS_PATH. getconf also recognizes the variables LOGNAME_MAX, PAGE_SIZE, PAGESIZE, and PASS_MAX.

Note: All variable names can be specified with or without a leading underscore (_).

ENVIRONMENT VARIABLES

The following environment variables affect the execution of getconf:

LANG Provides a default value for the internationalization variables that are unset or null. If LANG is unset or null, the corresponding default value from the internationalized environment is used. If one of the internationalization variables contains an invalid setting, the command behaves as if none of the variables have been defined.

LC_ALL
If this variable has been assigned a value, i.e. it is not a null string, this value overrides the values of all the other internationalization variables.

LC_CTYPE
Determines the internationalized environment for the interpretation of byte strings as characters (e.g. single-byte characters as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determines the format and contents of error messages.

NLSPATH
Determines the position of message catalogs for the processing of LC_MESSAGES.

EXIT STATUS

The following exit values are returned:

0 The specified variable is valid and information about its current status was written to the standard output.

>0 An error occurred.

EXAMPLES

In this example, the value of {NGROUPS_MAX} is obtained:

```
getconf NGROUPS_MAX
```

In this example, the value of {NAME_MAX} is obtained for a specific directory:

```
getconf NAME_MAX /usr
```

This example shows how to deal more carefully with results that might be unspecified:

```
if value=$(getconf PATH_MAX /usr); then
        if [ "$value" = "undefined" ]; then
                echo PATH_MAX in /usr is infinite.
        else
                echo PATH_MAX in /usr is $value.
        fi
else
        echo Error in getconf.
fi
```

Note that the following calls in a C program could return different results:

```
sysconf(_SC_POSIX_C_BIND);
```

and:

```
system("getconf POSIX2_C_BIND");
```

The sysconf(3C) call returns a value that corresponds to the conditions when the program is either compiled or executed. The system(3S) call to getconf always returns a value that corresponds to the conditions when the program is executed.

SEE ALSO

pathconf(2), confstr(3C), sysconf(3C).

User's Reference Manual

getext(1)
VxFS

NAME

`getext -` display extent attributes

SYNOPSIS

`getext` [`-f`] [`-s`] *file* ...

DESCRIPTION

`getext` displays extent attribute information associated with a set of files.

OPTIONS

`-f` Do not print the file name. Makes output a bit easier to parse.

`-s` Do not print output for files which do not have fixed extent sizes or reservation.

OUTPUT

```
file1:  Bsize  1024    Reserve 36    Extent Size  3  align noextend
```

The above line indicates a file with 36 blocks of reservation, a fixed extent size of 3 blocks, all extents aligned to 3 block boundaries, and the file unable to be grown once the current reservation is exhausted. The file system block size is 1024. Reservation and fixed extent sizes are allocated in multiples of the file system block size.

SEE ALSO

`setext`(1), `vxfsio`(7).

User's Reference Manual

getopt(1)

NAME

`getopt` - scan command line arguments for options

SYNOPSIS

`getopt` *optstring* `$*`

DESCRIPTION

`getopt` is being replaced by the Bourne shell built-in `getopts(1)` and will not be supported in future releases. A conversion tool `/usr/lib/getoptcvt` has been provided to convert earlier shell scripts from `getopt` to `getopts` [see `getoptcvt(1)`].

`getopt` reads the contents of the shell parameter `$*` and compares them with the `getopt` command line argument *optstring*. On the basis of this comparison, `getopt` either puts the contents of the shell parameter in a standard format and writes them to standard output in that form, or sends error messages to standard error.

The `getopt` command can be used in shell scripts to parse the arguments specified when the script is invoked. `getopt` breaks up the specified command line arguments into options and arguments to be passed to the script. The arguments can thus be easily parsed and checked for legal options.

OPERANDS

optstring
: A string that may comprise letters and colons. `getopt` treats the letters specified in *optstring* as legal options of the shell script. If a letter is followed by a colon, `getopt` expects the option to take an argument.

`$*`
: Shell parameter containing all the command line arguments of the shell script.

Parsing command line script arguments

When a shell script is invoked, the shell passes the name of the script to positional parameter `$0` and all the command line script arguments to the shell parameter `$*` [see `sh(1)`]. If the script includes a `getopt` command, `getopt` reads the contents of the shell parameter `$*` (see arguments to the script) and compares them with *optstring*, parsing the argument list into

- options
- arguments associated with an option
- other arguments.

`getopt` has the following rules for the writing of command line script arguments:

- Options may appear in any order.
- Options may be listed individually, separated by a blank and a minus sign, or grouped together with no blanks and only one minus sign at the start, e.g.

```
script –a –b –c
script –ab –c
script –abc
script –b –ac
```

- If an option has to have an argument, the argument can either come immediately after the option or be separated from it by a blank or a tab. It must be separated from the next option by a blank or a tab:

```
script –a –b arg –c
script –ab arg –c
script –abarg –c
script –acbarg
```

Caution:

`getopt` does not correctly handle a quoted group of arguments following an option and a blank. For example:

```
proc –a –b –o "xxx z yy" file
```

The argument `"xxx z yy"` to the `–o` option is not interpreted correctly. In such cases you should use the `getopts(1)` command instead of `getopt`.

If the command line options and arguments match the defined *optstring* in the `getopt` command, `getopt` writes them to standard output in a standard format. The options are listed separately, each preceded by a minus sign and followed by a blank. The argument for an option comes after the option and is separated from it by a blank. The end of the option list is marked by two minus signs (—); then come the other arguments, with blanks between them.

Example:

- *optstring*: `a:b:cdef`
- command line arguments: `–fb arg –caarg –de file1 file2`
- output from `getopt`: `–f –b arg –c –a arg –d –e –– file1 file2`

  See also *Example 1*.

If there is an illegal option in the argument list of a shell script, or if `getopt` cannot find a suitable argument for an option which has to have one, `getopt` issues an error message to that effect (see **ERROR MESSAGES**). If an argument has been assigned to an option which does not take one, `getopt` either tries to interpret it as an option or assumes that it has reached the end of the option list.

Example:

- *optstring*: `abc`
- command line arguments: `–a –barg –c`
- output from `getopt` if the `GETOPT_PREFIX` environment variable is not set:

  ```
  getopt: illegal option –– r
  getopt: illegal option –– g
  ```

  or:

  output from `getopt` if the `GETOPT_PREFIX` environment variable is set to *program*:

  ```
  program: illegal option –– r
  program: illegal option –– g
  ```

- *optstring*: `abc`
- command line arguments: `–a –b ––arg –c`
- output from `getopt`: `–a –b –– arg –c`

You can also use the special option –– to mark the end of the option list explicitly when you call a shell script. You only really need to do so if an argument which is not associated with an option is to start with a minus sign (see *Example 1*).

You can use the Bourne shell built-in `set(1)` to manipulate the parsed argument list that `getopt` writes to standard output: the command line

```
set –– `getopt ...`
```

passes the output of `getopt` to the shell parameter `$*`, thereby substituting the parsed command line arguments for the original contents of `$*` (the non-parsed command line script arguments; see also *Example 2*).

ERROR MESSAGES

If the `GETOPT_PREFIX` environment variable is not set:

```
getopt: option requires an argument ––x
```

if the GETOPT_PREFIX environment variable is set to *program*:

program: option requires an argument --x

-x has been identified in the getopt call as an option with arguments; however, you have not specified an argument for -x when calling the shell script that contains this getopt call.

If the GETOPT_PREFIX environment variable is not set:

getopt: option requires an argument --x

if the GETOPT_PREFIX environment variable is set to *program*:

program: option requires an argument --x

-y is not included in the list of legal options (*optstring*), but has been specified in the command line of the shell script.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The shell script proc has the following contents:

```
echo $*
getopt -abo:f $*
echo $*
...
```

The getopt line specifies that the legal options for proc are -a, -b, -o and -f and that option -o must have an argument.

You now invoke the script:

```
$ proc -bf -oarg file
-bf -oarg file
-b -f -o arg -- file
-bf -oarg file
```

The outputs from echo show that the command line arguments are passed to the shell parameter $* and that getopt does nothing to change this. As the argument list is acceptable, getopt does not issue any error messages, but simply writes the argument list to standard output in the standard format.

You now invoke the script with an illegal option:

```
$ proc -abx file
-abx file
getopt: illegal option -- x
-abx file
```

If you want the script to have an argument starting with a minus sign, you have to separate this argument from the option list with the double minus sign --:

```
$ proc -bf -oarg -- -argument
-bf -oarg -- -argument
-b -f -o arg -- -argument
-bf -oarg -- -argument
```

The special option -- marks the end of the option list; any arguments coming after it are no longer interpreted as options or as option-related arguments.

Example 2

The following code fragment from a shell script named `proc` shows how one might process the arguments of a shell script. The legal options for `proc` are `–a`, `–b`, and `–o`; `–o` requires an argument. Options `–a` and `–b` are mutually exclusive; if both are specified, only the one listed second applies:

```
GETOPT_PREFIX = proc
set –– `getopt abo: $*`
if [ $? != 0 ]
   then
      echo "usage: $0 [–a | –b] [–o <arg>]"
      exit 2
fi
for i in $*
   do
      case $i in
      –a | –b)        shift; FLAG=$i;;
      –o)             OARG=$2; shift 2;;
      ––)             shift; break;;
      esac
   done
```

This program code will accept any of the following `proc` calls as equivalent (among others):

```
proc –aoarg file1 file2
proc –a- o arg file1 file2
proc –oarg –a file1 file2
proc –boarg –a file1 file2
proc –a –o arg –– file1 file2
```

Next `proc` is invoked to show how this fragment of coding operates.

Suppose you invoke `proc` with the following command:

```
$ proc –boarg –a file1 file2
```

`getopt` establishes that the argument list is acceptable and passes it to `set` in the following format:

```
–b –o arg –a –– file1 file2
```

`set` successively passes the elements of this list to positional parameters `$1`, `$2` and so on, and passes the entire list to the shell parameter `$*`. The resultant values are:

```
$1 = –b
$2 = –o
$3 = arg
$4 = –a
$5 = ––
$6 = file1
$7 = file2
$* = –b –o arg –a –– file1 file2
```

As the argument list in your invocation of `proc` is acceptable, the condition in the `if` construct is not met, and the script then runs through the `for` loop.

First pass through the `for` loop:

The value of `$i` is `–b`. The `case` construction matches `–b` and assigns it to the shell variable `FLAG`. Following the `shift` function, positional parameters `$1`, `$2`, ..., `$*` have the following values:

```
$1 = –o
$2 = arg
$3 = –a
$4 = ––
$5 = file1
$6 = file2
```

```
$* = –b –o arg –a –– file1 file2
```

Second pass through the `for` loop:

The value of `$i` is `–o`, so the script takes the second branch of the `case` construction. On account of the `shift` command in the first pass of the `for` loop, the value of `$2` is `arg`; hence the assignment `OARG=$2` gives the shell variable `OARG` the value of the argument for the `–o` option. Following the `shift 2` command, the positional parameters have the following values:

```
$1 = –a
$2 = ––
$3 = file1
$4 = file2
$* = –b –o arg –a –– file1 file2
```

Third pass through the `for` loop:

The value of `$i` is now `arg`. As there is no branch of `case` for `arg`, nothing happens, and the values of the positional parameters stay as they are.

Fourth pass through the `for` loop:

Now the value of `$i` is `–a`. The `case` construction matches `–a` and assigns it to the shell variable `FLAG`, overwriting its old value of `–b`: options `–a` and `–b` are mutually exclusive, so only the second one that you specify is accepted, in this case `–a`. The values of the remaining parameters after the `shift` command are:

```
$1 = ––
$2 = file1
$3 = file2
$* = –b –o arg –a –– file1 file2
```

Fifth pass through the `for` loop:

The value of `$i` is `––`. The `break` command in the associated branch of the `case` construction causes the `for` loop to exit. Following the `shift` command, the values of parameters `$1`, `$2` and `$*` are:

```
$1 = file1
$2 = file2
$* = –b –o arg –a –– file1 file2
```

The script would then continue with further code for handling these arguments, and would then process them together with the options and their arguments.

SEE ALSO

`getoptcvt(1)`, `getopts(1)`, `sh(1)`, `getopt(3C)`.

User's Reference Manual

getoptcvt(1)

NAME

`getoptcvt` - convert `getopt` command calls to `getopts` command calls

SYNOPSIS

`/usr/lib/getoptcvt` [`–b`] *script*

DESCRIPTION

`getoptcvt` reads a shell script, converts it to use `getopts(1)` calls instead of `getopt(1)`, and writes the result to standard output.

OPTIONS

`–b` Makes the converted shell script *script* portable to earlier releases of the operating system. `/usr/lib/getoptcvt` modifies the script so that when the resulting shell script is executed, it determines at runtime whether to invoke `getopts` or `getopt`.

`–b` not specified:

The converted shell script will always use `getopts`.

script

The script file to be converted. The shell script must contain correct and meaningful `getopt` calls such as `set␣--␣`getopt␣...␣$*`. Otherwise, a "?" (question mark) is displayed, and the conversion program exits with an error.

EXAMPLES

All calls to the `getopt` command in the shell script `oldscript` are to be converted to `getopts` calls. The converted shell script is to be saved in the file `newscript`.

```
$ /usr/lib/getoptcvt oldscript > newscript
```

SEE ALSO

`getopt(1)`, `getopts(1)`.

User's Reference Manual

getopts(1)

NAME

getopts - scan command line arguments for options (get options)

SYNOPSIS

getopts *optstring name* [*arg ...*]

DESCRIPTION

The shell built-in getopts is used by shell scripts to parse arguments and options in the command line and to check for valid options. It supports all applicable rules of the command syntax standard.

getopts replaces the earlier getopt(1) command, which will not be supported in future releases of the operating system. It is therefore recommended that you always use getopts in place of getopt from now on. Existing shell scripts can be converted to use getopts calls instead of getopt with the conversion utility getoptcvt(1) .

getopts can be used in shell scripts to parse the arguments supplied when the script is invoked. The individual options and arguments are placed sequentially in shell variables and can thus be easily read and checked. If an illegal option is encountered in the argument list, or if getopt does not find a corresponding argument for an option that requires one, an appropriate error message is printed.

In order to ensure that the argument list is uniformly processed by all scripts and commands, getopts should always be used to parse the argument list and check for valid options.

OPERANDS

optstring

A string that may comprise letters and colons. getopts treats the letters specified in *optstring* as legal options of the shell script. If a letter is followed by a colon, getopts expects the option to take an argument or group of arguments, which must be separated from it by white space (blanks and tabs).

name Name of the shell variable in which getopts places the next option each time it is invoked.

arg ...

Argument list parsed by getopts.

arg not specified:

getopts parses the argument list of the command line with which the script was invoked.

MODE OF OPERATION

Each time it is invoked, getopts places the next option in the shell variable *name* and the index of the next argument to be processed in the shell variable OPTIND. Whenever the shell command interpreter or a shell script is invoked, OPTIND is initialized to 1.

When an option requires an argument, getopts places the argument in the shell variable OPTARG. Options that require an argument must be identified by a colon in *optstring*.

If an illegal option is encountered, a ? (question mark) is placed in the shell variable *name*. Illegal options are those which do not appear in *optstring*.

When the end of the option list is encountered, getopts exits with a non-zero exit status. The special option –- may be also used to identify the end of the options.

Changing the value of the shell variable OPTIND or invoking getopts with different sets of arguments may lead to unexpected results.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise

undefined or null, the system acts as if it were not internationalized.

EXAMPLES

The following fragment of a shell script `scrpt` shows how you might process the arguments for a script that can take the options `–a` or `–b`, as well as the option `–o`, which requires an option argument. Options `–a` and `–b` are mutually exclusive; if both are specified, only the one listed second applies:

```
while getopts abo: c
do
      case $c in
      a | b)          FLAG=$c;;
      o)              OARG=$OPTARG;;
      \?)             echo "usage: $0 [-a | -b] [-o <arg>]"
                      exit 2;;
      esac
done
shift `expr $OPTIND - 1`
```

This code accepts any of the following invocations of `scrpt` as equivalent:

```
scrpt -a -b -o "xxx z yy" file
scrpt -a -b -o "xxx z yy" -- file
scrpt -ab -o xxx,z,yy file
scrpt -ab -o "xxx z yy" file
scrpt -o xxx,z,yy -b -a file
```

NOTES

`getopts` exists both as an external command (`/usr/bin/getopts`) and as a built-in shell command, both in the Korn shell `ksh(1)` as well as in the Bourne shell `sh(1)`. The shell generates a new process to execute `/usr/bin/getopts`.

Some differences in behavior may occur when using `getopts`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`getopt(1)`, `getoptcvt(1)`, `ksh(1)`, `sh(1)`.

User's Reference Manual

gettxt(1)

NAME

`gettxt` - retrieve a text string from a message database (get text)

SYNOPSIS

`gettxt` *msgfile*`:`*msgnum* [`"`*default_msg*`"`]

DESCRIPTION

`gettxt` retrieves a text string associated with a message number from a message text file.

OPERANDS

msgfile

Basename (i.e. without the path prefix) of the message file from which `gettxt` is to retrieve *msgnum*. The basename can be up to 14 characters in length, but may not contain any of the following characters: \0 (null byte), / (slash), or : (colon).

msgfile must have been created by the `mkmsgs` utility and must reside in the directory `/usr/lib/locale/`*Locale*`/LC_MESSAGES`. The basename of the directory `/usr/lib/locale/`*Locale* corresponds to the language in which the text strings are written in the message file [see `setlocale`(3C)].

You can control which `/usr/lib/locale/`*Locale* directory is to be searched to find the message file containing the specified *msgnum* by setting the environment variable `LC_MESSAGES` (see below).

msgnum

Sequence number of the string to be retrieved from *msgfile*. The strings in *msgfile* are numbered sequentially from 1 to n, where n is the number of strings contained in *msgfile*.

`"`*default_msg*`"`

If `gettxt` cannot locate the required string in the message file whose pathname is defined by the `LC_MESSAGES` variable, it tries to retrieve the string associated with *msgnum* from the file `/usr/lib/locale/C/LC_MESSAGES/`*msgfile*. If the required string cannot be found there either, the message "Message not found!!" is issued. If either *msgfile* or *msgnum* does not exist, `gettxt` issues the specified *default_msg*.

"*default_msg*" not specified or "" (null):

If `gettxt` cannot locate the required string in the message file whose pathname is defined by the `LC_MESSAGES` variable, it tries to retrieve the string associated with *msgnum* from the file `/usr/lib/locale/C/LC_MESSAGES/`*msgfile*. If the required string cannot be found there either, or if *msgfile* does not exist, the string "Message not found!!" is displayed.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_TYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES` or `LC_TYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Let us assume the following files exist:

Contents of `input_file1`:

```
Hallo !\n\007
Gute   ... Nacht !\n
Guten Morgen !\n
```

Contents of `input_file2`:

```
Salut !\n
Bonne nuit !\n
```

The message files are created and installed as described below:

The directories `/usr/lib/locale/german/LC_MESSAGES` and `/usr/lib/locale/french/LC_MESSAGES` do not exist. The system administrator therefore creates the directory `/usr/lib/locale/german` with a file named `hallo` in it by issuing the following call:

```
$ mkmsgs -i german input_file1 hallo
```

The call

```
$ mkmsgs input_file2 hallo
```

creates a file named `hallo` in the current directory. To enable the subsequent use of this file in the `french` locale with `gettxt`, the system administrator must then perform the following actions:

```
$ mkdir /usr/lib/locale/french
$ mkdir /usr/lib/locale/french/LC_MESSAGES
$ cp hallo /usr/lib/locale/french/LC_MESSAGES/hallo
```

These files are accessed as follows: if `LC_MESSAGES` has a null value and the value of `LANG` is `german`:

```
$ gettxt hallo:1 "Hello!\n"
Hallo !
```

The terminal bell rings at the same time.

```
$ gettxt hallo:2 "Good Night!\n"
Gute   ... Nacht !
$ gettxt hallo:3 "Good Morning!\n"
Guten Morgen !
$ LC_MESSAGES=french
$ export LC_MESSAGES
$ gettxt hallo:1 "Hello!\n"
Salut !
$ gettxt hallo:2 "Good Night!\n"
Bonne nuit !\n
$ gettxt hallo:3 "Good Morning!\n"
Message not found!!
```

Invoking `hallo:3` causes the message `Message not found!!` to be displayed, since there are only two messages in the file `/usr/lib/locale/french/LC_MESSAGES/hallo`.

```
$ LC_MESSAGES=invalid
$ gettxt hallo:1 "Hello!\n"
Hello
$ gettxt hallo:2 "Good Night!\n"
Good Night!
```

Invoking `hallo:1` or `hallo:2` causes the default message "*default_msg*" to be displayed, as there is no existing file named `/usr/lib/locale/invalid/LC_MESSAGES/hallo`. If no "*default_msg*" were specified, the message `"Message not found!!"` would be displayed [see `gettxt`(3C)].

FILES

`/usr/lib/locale/`*Locale*`/LC_MESSAGES/*`
: Message files created by `mkmsgs` for different languages.

`/usr/lib/locale/C/LC_MESSAGES/*`
: Default message files created by `mkmsgs`.

SEE ALSO

`exstr`(1), `mkmsgs`(1), `srchtxt`(1), `gettxt`(3C), `setlocale`(3C).

User's Reference Manual

gettypes(1)

NAME

`gettypes -` displays SCSI devices and some of their parameters

SYNOPSIS

`/sbin/gettypes`

`/sbin/gettypes –R` *class*

`/sbin/gettypes –A` *class*

DESCRIPTION

Using `gettypes`, the SCSI devices that were released as part of the current release can be displayed, as well as the devices currently configured on the system. Particular device types can be selected using the classes that are to be specified.

OPTIONS

`–R` SCSI devices for current release.

`–A` SCSI devices for the current configuration.

class

Device type. The following options are possible for *class*:

`cdrom`, `dat`, `disk`, `floppy`, `jukebox`, `raid`, `rod`, `smc`, `tape`, `videotape`, `worm`, `all`.

OUTPUT

Output with the –R option

In the case of `disk`|`cdrom`|`worm`|`rod`:

type:*name*:*capacity*:*number_cylinders*:*capacity/cylinder*:*codeno._writecache*: \
codeno._slotwidth :*codeno._transmission_width_of_data*[:*codeno._olr*]

Unknown devices of type `disk`, `rod`, `worm` and `cdrom` that are connected temporarily are also displayed (but only if invoked with superuser rights).

In the case of `videotape`|`dat`|`smc`|`tape`:

type:*name*:*capacity_minimum*:*capacity_maximum*:*options*: \
codeno._slotwidth[:*codeno._changer*][:*codeno._olr*]

In the case of `raid`:

type:*name*:*capacity*:*codeno._slotwidth*

In the case of `floppy`:
type:*name*:*capacity_minimum*:*capacity_maximum*:*codeno._slotwidth*

In the case of `jukebox`:

type:*name*:*slots_minimum*:*slots_maximum*:*drives_minimum*:*drives_maximum*: \
drive_type:*codeno._slotwidth*

The individual fields have the following meaning here:

type Device type: `cdrom`, `dat`, `disk`, `floppy`, `jukebox`, `raid`, `rod`, `smc`, `tape`, `videotape`, `worm`.
The type `jukebox` is output for jukebox.

name The name of the device as it appears with the `autoconf(8)` command.

capacity

Numeric value that can be interpreted as follows:

`disk` Numeric value in megabytes.

`cdrom|worm|rod`
Maximum capacity; numeric value in megabytes.

`videotape|dat|smc|tape`
Numeric value in megabytes.

`raid` Always digit `0`. No capacity is specified.

`floppy`
Numeric value in kilobytes.

number_cylinders
Numeric value describing the number of cylinders.

capacity_minimum/capacity_maximum
The capacity on the cassette (numeric value in megabytes), which can move between this minimum or maximum sizes depending on the variant (design).

capacity/cylinder
Number of bytes per cylinder.

codeno._writecache
`0` Cannot be enabled.

`1` Can be enabled.

codeno._slotwidth
`0` Stand alone device.

`1` Dual width = 5 ¼ inch device.

`2` Single width = 3 ½ inch device.

codeno._olr
`OLR="y"`
Online replacement possible

`OLR="n"`
Online replacement impossible

None Online replacement impossible

slots_minimum/slots_maximum
Number of slots that can vary within these limits depending on the variant (design).

codeno._transmission_width_of_data
`8` 8-bit wide data transmission.

`16` 16-bit wide data transmission.

options
Possible options separated by a blank, e.g. "`n o h c`".

codeno._changer
If the drive has an automatic changer option (this does not mean a jukebox as a stand alone device!), a corresponding code number is added: ":changer="true";slots_min="9";slots_max="9"". The digit "9" stands for the actual number of slots.

drives_minimum/drives_maximum
Number of drives that may vary within these limits depending on the variant (design).

drive_type
The drive type, e.g. "`rod`", is specified here for the jukebox.

Output with the –A option

Output for the `–A` option is displayed as follows: the name of the device as it appears in the `autoconf(8)` command, devices class, maximum capacity (where possible), the pathname plus appropriate options (in the

case of device class, `tape`), and the alias name (where available).

Output without option

If `gettypes` is called without specifying an option, a formatted list of all devices currently configured on the system is displayed. The information contained in the output is the same as that displayed when `gettypes` is called with the "`–A all`" option.

EXAMPLES

When `gettypes` is called without specifying an option the following list is displayed:

```
Alias        Name     Type       Size      Path
-------------------------------------------------------------------------
disk0000     MP12     disk       4023  MB  /dev/ios0/rsdisk000
disk0001     MP12     disk       4023  MB  /dev/ios0/rsdisk001
-------      OS25     cdrom      650   MB  /dev/ios0/rsdisk006
disk0044     MP81.D   disk       1005  MB  /dev/ios0/rsdisk044
smc0         MC12     smc        525   MB  /dev/ios0/rstape040[n][o][v]
floppy0      FD11     floppy     1440  KB  /dev/ios0/rsfdisk005
```

Device type **disks**

`gettypes -A disk` gives:

```
MP12 disk 4023 MB /dev/ios0/rsdisk000 disk0000
MP12 disk 4023 MB /dev/ios0/rsdisk001 disk0001
MP81.D disk 1005 MB /dev/ios0/rsdisk044 disk0044
```

`gettypes -R disk` gives:

```
disk:CDC94181_702:572:1532:391680:0:1:8
disk:MIC1578:315:1197:276480:0:1:8
disk:MIC1588:635:1607:414720:0:1:8
disk:MIC1684:323:1753:193536:0:1:8
disk:MP12:4023:8047:524288:1:2:8
disk:MP14:2001:4002:524288:1:2:8
disk:MP14-85:2046:2626:817152:0:2:8
disk:MP18:322:786:430080:0:1:8
disk:MP19:1014:2028:524288:0:2:8
disk:MP24:2001:4002:524288:1:2:16
disk:MP25:4023:8047:524288:1:2:16
disk:MP26:2001:4002:524288:0:2:8
disk:MP29:4023:8047:524288:1:2:8
disk:MP33:400:1425:294912:0:2:8
disk:MP34:585:1537:399360:0:1:8
disk:MP50:1004:1853:568320:0:2:8
disk:MP64:1274:2175:614400:0:1:8
disk:MP65:1278:2078:645120:0:1:8
disk:MP67:1001:1644:638976:0:2:8
disk:MP81:1005:2692:391680:0:2:8
disk:MP81.D:1005:2692:391680:0:2:8
disk:MP85:2046:2626:817152:0:2:8
disk:MP88:2774:2733:1064448:0:1:8
disk:MP90:522:3302:165888:0:2:8
disk:SG1581:500:1459:359424:0:2:8
disk:SEAGATE ST15150N:4023:8047:524288:0:2:8
disk:TOSHIBA CD-ROM XM-3501TA:288:147936:2048:0:2:8
disk:QUANTUM EMPIRE_1080S*SNI:1005:2692:391680:0:2:8
```

Device type **tapes**

`gettypes -R dat` gives:

```
dat:MC23:4096:8000:n o h c:2
```

```
dat:MC42:4096:68820:n o h c:1:changer="true";slots_min="1";slots_max="1"
```

`gettypes -R tape` gives:

```
tape:DLT 2000:10000:30000:n o h c:0
tape:DLT 4700:10000:280000:n o h c:0
tape:HP 88780:10:150:n o h:0
tape:PXT0-901:800:1600:n o c:0
tape:MB11:10:150:n o h:0
tape:PXT0-90X:800:1600:n o c:0
tape:PXT0-911:800:1600:n o c:0
tape:PXT0-91A:800:1600:n o c:0
tape:PXT0-922:800:1600:n o c:0
tape:PXT0-92A:800:1600:n o c:0
tape:PXT1-B11:10000:100000:n o c:0
tape:PXT1-B1A:10000:10000:n o c:0
tape:STK 9490:800:2500:n o c:0
```

Device type **jukeboxes**

`gettypes -R jukebox` gives:

```
jukebox:9083-10 (Jukebox):16:16:1:1:rod:0
jukebox:9083-102 (Jukebox):16:16:1:1:rod:0
jukebox:9083-10x (Jukebox):144:144:2:4:rod:0
jukebox:9083-10x2 (Jukebox):144:144:2:4:rod:0
jukebox:9083-20 (Jukebox):32:32:2:2:rod:0
jukebox:9083-202 (Jukebox):32:32:2:2:rod:0
jukebox:9083-6x (Jukebox):88:88:2:4:rod:0
jukebox:9083-6x2 (Jukebox):88:88:2:4:rod:0
jukebox:9084-10 (Jukebox):10:10:1:1:videotape:0
jukebox:9084-1x (Jukebox):11:11:1:2:videotape:0
jukebox:9084-2x (Jukebox):40:40:1:4:videotape:0
jukebox:9084-2y (Jukebox):80:80:1:4:videotape:0
jukebox:9084-3x (Jukebox):20:20:1:2:videotape:0
jukebox:ARL4-xx (Jukebox):22:131:2:5:worm:0
jukebox:EXB-120 (Jukebox):116:116:1:4:videotape:0
jukebox:PXM1-OS35 (Jukebox):238:238:6:6:rod:0
jukebox:PXM1-OS4x (Jukebox):32:76:2:4:rod:0
jukebox:PXM1-OS4y (Jukebox):128:238:4:12:rod:0
jukebox:PXMO-OS3x (Jukebox):32:76:2:4:rod:0
jukebox:PXT4-MCxx (Jukebox):18:18:1:2:dat:0
```

FILES

`/etc/default/diskinfo/setinfo/`*file*

`/etc/default/tapeinfo/setinfo/`*file*

`/etc/default/floppyinfo/setinfo/`*file*

`/etc/default/raidinfo/setinfo/`*file*

`/etc/default/jukeboxinfo/setinfo/`*file*

`/etc/default/tapechangerinfo/setinfo/`*file*

`/etc/device.tab`

SEE ALSO

`autoconf`(8).

grep(1)

NAME

grep - global regular expression print

SYNOPSIS

grep [*option ...*] [*patternlist*] [*file ...*]

DESCRIPTION

grep reads lines from one or more files or from standard input and compares these lines with the specified patterns. Unless told otherwise (by options), grep copies every line that matches one of the patterns to standard output.

grep permits the use of simple regular expressions in the specified pattern [see expressions(5)].

If you specify more than one input file, the relevant file name will be displayed before each output line.

The pattern with which grep is to compare the individual lines are specified either using *patternlist* or via the options –e*patternlist* or –f *patternfile*. At least one of these three arguments must be specified, multiple specifications are allowed. Multiple specifications are ORed.

OPTIONS

The options –E and –F are mutually exclusive, as are the options –c, –l, and –q.

No option specified:
grep outputs all lines that match at least one of the patterns specified in *patternlist*. If you specify more than one input file, each output line will be preceded by the name of the file in which the line was found.

–E (E – extended) You can specify extended regular expressions as patterns. grep –E behaves in a similar way to egrep.

–F (F – fixed) All patterns are interpreted as character strings. grep –F behaves in a similar way to fgrep.

–b (b – block) Each output line is preceded by the number of the block in which it was found.

Each file is made up of 512-byte blocks which are numbered consecutively from 0.

The –b option is sometimes useful in locating disk block numbers by context (see the *offset* argument for the od(1) command, for example).

–c (c – count) grep outputs only the number of lines found (i.e. the lines that grep would have displayed without the –c option, see *Example 3*); the lines themselves are not displayed.

–e *patternlist*
(e – expression) Synonymous with *patternlist*. The –e option must always be specified if the first expression in *patternlist* begins with a dash (–). When combined with –e, such a pattern list is not interpreted as an option, but as a list of patterns with which grep is to match the individual lines. Multiple specifications of the –e option are allowed.

–f *patternfile*
(f – file) grep reads the pattern list from the *patternfile* file. Each line in *patternfile* is interpreted as a simple regular expression. Multiple specifications of the –f option are allowed.

–h (h – hidden) When searching multiple files, grep does not write the file name before each output line.

–i (i – ignore) grep does not distinguish between uppercase and lowercase.

–l (l – list) grep simply outputs the names of files that contain at least one of the matching lines. (These are the lines that egrep would output if the –l option were omitted, see *Example 4*.) Each file name is printed just once. The lines themselves are not displayed.

`-n` (n – number lines) Each output line is preceded by its line number in the relevant input file. Line numbering starts at 1. If `grep` is reading from standard input, the line number refers to the standard input.

`-q` (q – quiet) `grep` does not produce any output, but returns only the corresponding exit status.

`-r` (r – recursive) Names that refer to directories are processed recursively; in other words, all the files and subdirectories in that directory are scanned as well.

`-s` (s – silent) Error messages produced for non-existent files or files which the user is not authorized to read are suppressed.

`-v` (v – vice versa) `grep` outputs all lines that *do not* match the specified pattern.

In conjunction with option `-c`: `grep` prints only the number of lines that do not match.

In conjunction with option `-l`: `grep` only outputs the names of files containing such lines.

In conjunction with option `-q`: `grep` returns only the corresponding exit status.

`-x` (x – exact) `grep` does not output lines containing one of the specified patterns and no other characters.

In conjunction with option `-c`: `grep` prints only the number of such lines.

In conjunction with option `-l`: `grep` only outputs the names of files containing such lines.

In conjunction with option `-q`: `grep` returns only the corresponding exit status.

`-y` `grep` does not distinguish between uppercase and lowercase.

`--` End of the list of options. Must be specified if the first argument begins with `-`.

patternlist

List of simple regular expressions that `grep` is to use in searching for matching input lines [see `expressions`(5)]. Individual regular expressions must be separated by the newline character. A newline character within *patternlist* functions like an OR separator.

If *patternlist* contains newline characters or other characters that have a special meaning for the shell, you must enclose the specified *patternlist* in single quotes: '*patternlist*'.

file Name of the file that `grep` is to scan. You may name any number of files.

file not specified: `grep` reads its input from standard input.

grep, fgrep and egrep

The `grep`, `fgrep` and `egrep` commands perform similar functions and are largely identical in terms of usage. The following section lists the most important differences between these commands.

`grep` processes simple regular expressions if the options `-E` or `-F` are not specified.

`fgrep` only processes character strings. `fgrep` can search for a large number of character strings very efficiently: `fgrep` searches each individual line for all of the character strings. `fgrep` behaves in a similar way to `grep -F`.

`egrep` processes simple regular expressions. These include simple regular expressions with the following exception: The simple regular expressions \(*regexpression*\) and \{...\} have no special meaning for extended regular expressions and are therefore not processed by `egrep`. `egrep` behaves in a similar way to `grep -E`.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

The `LC_COLLATE` environment variable governs the collating sequence.

`LC_TYPE` governs character classes and character conversion (shifting, option `-i`/`-y`).

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other

environment variables that affect internationalization.

EXIT STATUS

`0` Lines found

`1` No line found

`>1` Syntax error or "Cannot open file". This exit status remains valid even if lines have been found in other input files.

EXAMPLES

The files `customer1` and `customer2` will be used as a basis for all following examples. Their contents are given below:

`customer1`:

```
080685    999.98  20 Units  Item  038   Johnson Ltd.
120387   1240.25   3 Units  Item  023   Skinner Ltd.
180588    330.87   1 Units  Item  332   Skinner Ltd.
```

`customer2`:

```
skinner robert, 16 garden hill, london ec3
```

Example 1

Output lines that match a specified pattern (without an option and with option `-i`):

```
$ grep Skinner customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:180588    330.87   1 Units  Item  332   Skinner Ltd.
```

If you also wish to display lines containing the word `skinner` in lowercase you enter:

```
$ grep -i skinner customer1 customer2
customer1:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:180588    330.87   1 Units  Item  332   Skinner Ltd.
customer2:skinner robert, 16 garden hill, london ec3
```

More complicated patterns can be set up with the help of regular expressions, e.g.:

Display all 1987 entries from the file `customer1`; these are lines that contain the number 87 in columns 5 and 6:

```
$ grep '^....87' customer1
120387   1240.25   3 Units  Item  023   Skinner Ltd.
```

Example 2

Output lines that *do not* match the specified pattern (option `-v`):

```
$ grep -v '^1' customer1 customer2
customer1:080685    999.98  20 Units  Item  038   Johnson Ltd.
customer2:morrow lance, 86 sherwood street, london w1
customer2:skinner robert, 16 garden hill, london ec3
```

Example 3

Display the number of found lines (option `-c`):

First display the number of lines that start with 1 for each input file.

```
$ grep -c '^1' customer1 customer2
customer1:2
customer2:0
```

The number of lines that *do not* start with 1 are to be displayed next.

```
$ grep -c -v '^1' customer1 customer2
customer1:1
customer2:2
```

Example 4

Display file names only (option `-l`):

The names of files containing lines that begin with a 1 are to be output first.

```
$ grep -l '^1' customer1 customer2
customer1
```

The names of files containing lines that do not start with 1 are displayed next.

```
$ grep -l -v '^1' customer1 customer2
customer1
customer2
```

Example 5

Display found lines with their line numbers (option `-n`):

```
$ grep -n -i skinner customer1 customer2
customer1:2:120387   1240.25   3 Units  Item  023   Skinner Ltd.
customer1:3:180588    330.87   1 Units  Item  332   Skinner Ltd.
customer2:2:skinner robert, 16 garden hill, london ec3
```

SEE ALSO

ed(1), egrep(1), fgrep(1), sed(1), sh(1), stdio(3S), expressions(5).

Programmer's Guide, *Internationalization – Localization*.

User's Reference Manual

groups(1)

NAME

`groups` - print group membership of user

SYNOPSIS

`groups` [*login_name*]

DESCRIPTION

`groups` lists the groups to which a user belongs on standard output.

OPERANDS

login_name

Login name of the user whose group membership is to be checked. The groups to which the specified user belongs are listed on standard output.

login_name not specified:

The groups of the user logged in at the terminal are displayed.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

User John wishes to check his own user group:

```
$ groups
group1
```

Example 2

To check the group his colleague Anne belongs to, John must enter her login name.

```
$ groups anne
group2
```

FILES

`/etc/passwd`

The `/etc/passwd` file contains the login name and the group ID of each user.

`/etc/group`

The `/etc/group` file contains a list of all user groups that have been set up. Each user belongs to the group specified in `/etc/passwd` and possibly to other groups as specified in `/etc/group`.

SEE ALSO

`group(4)`, `passwd(4)`.

User's Reference Manual

groups(1-ucb)
BSD Compatibility Package

NAME

`groups` - display a user's group memberships

SYNOPSIS

`/usr/ucb/groups` [*user* ...]

DESCRIPTION

With no arguments, `groups` displays the groups to which you belong; else it displays the groups to which the *user* belongs. Each user belongs to a group specified in the password file `/etc/passwd` and possibly to other groups as specified in the file `/etc/group`. If you do not own a file but belong to the group which it is owned by then you are granted group access to the file.

FILES

`/etc/passwd`
`/etc/group`

SEE ALSO

`getgroups`(2) .

hash(1)

NAME

`hash` - process shell hash table

SYNOPSIS

Format 1

`hash` [*name* ...]

Format 2

`hash –r`

DESCRIPTION

The Bourne shell built-in `hash` has two functions:

- it can write the contents of the hash table on standard output or enter the specified command in the hash table (**Format 1**),
- it can delete the contents of the hash table (**Format 2**).

Each shell maintains its own hash table, in which it enters all commands that are invoked under their basic file names (basenames). Whenever you invoke a command under its basename, the shell first searches the hash table for your command. This accelerates the search process [see `sh(1)`]. If the command is not yet in the hash table of the current shell, it is entered there.

When you start a subshell, its hash table is empty. When you terminate the subshell, the parent shell returns with its own hash table, and the hash table of the subshell is deleted.

In the Korn shell `ksh(1)` , `hash` is an exported alias for `alias –t –`; `alias(1)` is a Korn shell built-in command.

Format 1: Display or extend hash table

`hash` [*name* ...]

name Basename of a command, an executable shell script, or an executable program. The shell searches for this file by examining the contents of the `PATH` variable. When the file is found, the following information is passed to `hash`:

- the appropriate relative path name in the form `./`*name* if the current directory is assigned to the `PATH` variable and contains *name*, or
- the absolute path name.

The `hash` command enters this path name into the hash table. If *name* is not present in any of the directories assigned to the `PATH` variable, an error message is issued.

The following cannot be specified for *name*:

- shell scripts for which you have no execute permission.
- commands with a *name* that includes a slash. This means that you cannot enter absolute or relative path names.
- shell built-ins, since they are subroutines of `sh` or `ksh` and thus do not have names that represent executable files.

By the same token, executable shell scripts and commands are not entered in the hash table unless they are invoked under their basenames.

name not specified:

The `hash` command writes the contents of the hash table on the standard output. The output is structured as follows:

```
hits    cost    command
```

```
1       1       /bin/mail
1       2       /usr/bin/tar
1       1       /bin/ls
1       1       /bin/cat
1       1       /bin/chmod
```

The meanings of the entries in the above columns are given below:

`hits` The number of times the shell has executed this command since it was entered in the hash table. The digit 0 identifies commands that have so far only been entered, but not yet executed.

An asterisk immediately after the number identifies executable files that are not located in the default directories for `PATH`, i.e. are not contained in `/bin` or `/usr/bin`.

`cost` A measure of how many path names in the `PATH` variable were searched by the shell in order to find the specified command.

`command`
Absolute path name of the executable file, or a path name in the form `./`*name* for a file in the current directory.

Format 2: Delete hash table

`hash –r`

`–r` Deletes the contents of the hash table.

If you modify the value of the `PATH` variable, the contents of the hash table are automatically deleted.

When you terminate the current shell, the hash table associated with this shell is also deleted.

You should always delete the contents of the hash table in the current shell in the following circumstances:

You have created an executable file in a directory `dira` whose path name is assigned to the `PATH` variable. Some other directory `dirb`, also specified in `PATH`, already contains an identically named command file which already appears in the current hash table. In this case the shell will always execute the older command, even if its directory (`dirb`) comes after the first one (`dira`) in the `PATH` variable.

Deleting the hash table with `hash –r` will cause the shell to execute the first command it finds the next time either of these commands is invoked. In other words, the sequence of the entries in the `PATH` variable will be the determining factor. The path name of the first command found is now entered into the hash table, i.e. your command will always be executed from now on when you invoke it under its basename.

ERROR MESSAGES

name`: not found`

This error message may be produced for any of the following reasons:

- *name* is not contained in any of the directories whose respective paths have been entered in the `PATH` variable.
- *name* is actually contained in one of these directories, but is not executable.
- *name* is a shell built-in or a shell function.

ENVIRONMENT VARIABLES

`PATH` Search path of the shell

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

Example 1

Display the contents of the hash table:

```
$ echo PATH
/bin:/usr/bin:.
$ hash
hits    cost    command
1       1       /bin/mail
1       2       /usr/bin/tar
2       1       /bin/ls
3       1       /bin/cat
1*      3       ./stich
1       1       /bin/uname
```

Example 2

Add a command to the hash table:

```
$ hash lpr
$ hash
hits    cost    command
.
.
0       1       /bin/lpr
.
.
```

This means that the `/bin/lpr` command has just been entered in the hash table and has not yet been executed. It was found by the shell in the directory whose path name was specified first in the `PATH` variable.

Example 3

The following excerpt from a session demonstrates when the hash table is cleared:

```
$ hash
hits    cost    command
1       1       /bin/mail
1       2       /usr/bin/tar
2       1       /bin/ls
3       1       /bin/cat
0       1       /bin/lpr
1*      3       ./stich
1       1       /bin/uname
$ sh
$ hash
hits    cost    command
$ who
...
$ ls
...
$ hash
hits    cost    command
1       1       /bin/who
1       1       /bin/ls
$ CTRL-D
$ hash
hits    cost    command
1       1       /bin/mail
1       2       /usr/bin/tar
2       1       /bin/ls
3       1       /bin/cat
0       1       /bin/lpr
```

```
1*      3       ./stich
1       1       /bin/uname
1       1       /bin/sh
```

The hash table in the new subshell is empty. When the subshell is terminated, the hash table of the parent shell becomes valid again, while that of the subshell is deleted.

NOTES

`hash` exists both as an external command (`/usr/bin/hash`) and as a built-in shell command in the Bourne shell `sh(1)`. The shell generates a new process to execute `/usr/bin/hash`.

Some differences in behavior may occur when using `hash`, depending on which command is being used. The possible differences are not described specifically.

SEE ALSO

`alias(1)`, `sh(1)`.

User's Reference Manual

hashcheck(1)

NAME

`hashcheck` - check hashed spelling list

SYNOPSIS

`/usr/lib/spell/hashcheck` *word_list*

DESCRIPTION

`hashcheck`, `hashmake` and `spellin` are a group of auxiliary routines for the spell-checking program `spell`. The `spell` command uses spelling lists to determine the correct spelling of words. You can use the auxiliary routines to maintain and extend the existing spelling lists and to create new lists.

`hashcheck` reads the compressed spelling list *word_list* (created by `spellin`) and recreates the original nine-digit hash code (as created by `hashmake`). This code is written to standard output.

`hashcheck` takes as input the system files `hlista` or `hlistb`, or any other file created or modified by `spellin`.

By applying `hashcheck` to an existing compressed spelling list and `hashmake` to a file of selected words, you can compare the two output files to determine if the selected words are present in the existing spelling list.

SEE ALSO

`hashmake`(1), `spell`(1), `spellin`(1).

User's Reference Manual

hashmake(1)

NAME

`hashmake` - create hashed spelling list

SYNOPSIS

`/usr/lib/spell/hashmake`

DESCRIPTION

`hashmake`, `hashcheck` and `spellin` are a group of auxiliary routines for the spell-checking program `spell`. The `spell` command uses spelling lists to determine the correct spelling of words. You can use the auxiliary routines to maintain and extend the existing spelling lists and to create new lists.

`hashmake` reads a list of words from standard input and writes the coded words to standard output in the form of a hash list. The hashing technique generates a nine-digit hash code.

This is the first step in creating a new spelling list [see `spell(1)`] or adding new words to an existing list.

`hashmake` must always be used before `spellin`.

SEE ALSO

`hashcheck(1)`, `spell(1)`, `spellin(1)`.

User's Reference Manual

hd(1)

NAME

hd - hexadecimal dump

SYNOPSIS

hd [–*format* ...] [–A] [–t] [–s *offset*[*][wlbk]] [–n *count*[*][wlbk]] [*file* ...]

DESCRIPTION

hd prints the contents of files in hexadecimal, octal, or decimal notation or as character strings. It can also display the position of characters within a string.

If no format, offset, and count is specified:

The output is identical to that of hd –abx –A, i.e. addresses and bytes are shown in hexadecimal. In addition, all bytes that represent printable characters are shown as such, and non-printing characters are represented by a dot. Addresses are displayed at the beginning of each line, followed by the hexadecimal representation of the bytes in the next few columns and the actual characters (with dots if required) on the right.

Addresses are calculated relative to the start of the file. If no file is specified, the standard input is read; otherwise, the contents of all specified files are listed.

–*format*

Format that determines which individual byte blocks are to be interpreted and output.

A format consists of the following components:

- the byte block option (a, b, c, l or w) and
- the method by which a byte block is to be interpreted in the output, i.e.: hexadecimal (x), decimal (d) or octal (o).

All specified interpretation methods are applied to all specified byte blocks in a format. Format options can be combined and repeated to display addresses, characters, words, etc. in various ways. For example, you could combine –ax -bx into –abx, or specify –cxdo to show all characters in hexadecimal, decimal, and octal.

Byte block options

a (address) Format option for addresses. Addresses are only interpreted by one method, i.e. in hexadecimal, octal, or decimal. The address is always shown at the start of each line to be displayed, or in the first line of an output block if multiple lines are required for the formats.

b (byte) Format option for bytes.

c (character) Format option for characters. All printing characters are displayed. C-language escape sequences are output as defined in the language; all other characters are shown in octal, hexadecimal or decimal, depending on the interpretation method.

l (long word) Format option for 4 bytes.

w (word) Format option for 2 bytes.

Interpretation methods

x hd interprets addresses or byte blocks as hexadecimal numbers.

d hd interprets addresses or byte blocks as decimal numbers.

o hd interprets addresses or byte blocks as octal numbers.

Without an interpretation method, but with a byte block option:

The format is interpreted as –xdo.

With no byte block option except for addresses:

`hd` uses `–bx` in addition to the specified address format.

Without a byte block option, but with an interpretation method:

The format is interpreted as `–acbwl`.

–format not specified:

`hd` acts as if `hd –abx –A` were specified.

`–A` (ASCII) All printable characters appear unaltered; non-printing characters are represented by a dot. The characters are shown in the column to the right of the first output format.

`–t` (text) If this option is set, `hd` will ignore all format options that do not affect addresses. Each text line is printed with the address shown at the start of the line. Long lines are folded (split). The notation for control characters (values 0x00 to 0x1f) is a caret followed by the corresponding character (^@ to ^_). Bytes with the high order bit set are shown with a preceding tilde (~), but without the high order bit itself. The caret, tilde and backslash characters are preceded by a backslash in the output. Some special cases are represented by numeric values, e.g. a 7-bit DELETE (127) character as \177, and an 8-bit (255) as \377, respectively.

`–s` *offset*[`*`][`wlbk`]
Relative address at which printing of the file is to begin. If you do not specify a file or if you supply the input via a pipeline, a corresponding number of bytes will be skipped. `hd` will abort processing of the current file if you give it an invalid address.

The relative address consists of a number, which can be specified in decimal, hexadecimal (preceded by `0x`) or octal (preceded by `0`), and optionally a flag indicating a unit size directly after the number. The possible flags are:

`w` 2-byte units (i.e. one word)

`l` 4-byte units (i.e. one long word)

`b` 512-byte units (i.e. half of one Kbyte)

`k` 1024-byte units (i.e. one Kbyte)

You can mark the difference between a hexadecimal number that includes the digit `b` and the `b` flag by putting an asterisk (*) between the hexadecimal number and the `b` flag.

The following examples show how the offset can be specified:

`–s 111` (111 bytes), `–s 124l` (496 bytes), `–s 0xa*b` (5120 bytes), `–s 011k` (9216 bytes).

`–s` *offset*[`*`][`wlbk`] not specified:
The output begins at the start of the file.

`–n` *count*[`*`][`wlbk`]
Number of bytes to be dumped. The *count* is specified in the same format as the offset, i.e. in decimal, hexadecimal or octal, followed by an optional `w`, `l`, `b` or `k` flag (see the `–s` option).

file Name of the file to be dumped by `hd`. More than one file may be named.

file not specified:

`hd` reads input lines from standard input.

SEE ALSO

`od`(1) .

User's Reference Manual

head(1)

NAME

`head` - display opening lines of files

SYNOPSIS

`head` [–n *number*] [––] [*file* ...]

DESCRIPTION

`head` copies the opening lines of a file to standard output. If no file is given, `head` reads from standard input.

OPTIONS

–n *number*

Number of lines to be output.

–n not specified: The first 10 lines are output.

–n *number* is equivalent to the old –*number* option, which will continue to be supported.

–– If *file* begins with a dash (–), the end of the command-line options must be marked with ––.

file Name of the input file. If more than one file is named, the files will be processed in the order in which they are listed, and the output of each file begins with: ==>*file*<==.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

To see the first 5 lines of three files, you enter the `head` command as shown below:

```
$ head -n 5 file1 file2 file3
```

The first five lines of each of the three files are written to standard output as follows:

```
==>file1<==
Lines 1-5 of file 1
==>file2<==
Lines 1-5 of file 2
==>file3<==
Lines 1-5 of file 3
```

SEE ALSO

`cat`(1), `more`(1), `pg`(1), `tail`(1).

User's Reference Manual

ic(1)

NAME

`ic -` compile an international database (internationalizationcompiler)

SYNOPSIS

`ic` [*preprocessor_option ...*] [*option ...*] [*source*]

DESCRIPTION

`ic` takes a database source file as input and generates from it intermediate files which can be further processed by the system administrator to produce databases that comply with the 5.4 format.

If the `–VSINIX` option is set, `ic` behaves as under SINIX versions V5.22 and V5.23 and generates a binary international NLS database from the source file. `ic` reads the source text either from a file or from standard input. The output produced by `ic` is written to one or more files (see the `–o` option).

`ic` uses the C preprocessor of the C-DS C development system.

Preprocessor options

preprocessor_option can be: `–D`, `–U` or `–I`.

These options are passed on to the C preprocessor.

`–D`*name*[`=`*def*]
: The value *def* is assigned to the symbolic constant *name*. This option has the same effect as the preprocessor directive `#define`.

  `=`*def* not specified: *name* is assigned a value of 1.

`–U`*name*
: Any initial definitions that assign a value to the symbolic constant are removed. This option has the same effect as the preprocessor directive `#undef`.

`–I`*dir*
: The directory named *dir* is also searched for header files.

OPTIONS

No option specified:
: `ic` generates a number of intermediate files and a shell script which can be used by the system administrator to create a database in 5.4 format.

`–V5.4`
: This option is set by default. `ic` generates a number of files and a shell script which enables the system administrator to create a database in 5.4 format.

`–VSINIX`
: `ic` generates a binary NLS database in SINIX V5.2x format from the source file.

`–M` *modifier*
: Causes `ic` to use information defined by *modifier* when generating the temporary files. *modifier* is the parameter which you set as the `@`*modifier* element for the shell variable `LANG`.

The `–VSINIX` option may not be used in combination with either `–V5.4` or `–M`.

`–v` When the source file is successfully compiled, statistical information is output to standard error indicating:

- the number of simple and multi-byte characters
- the number and type (e.g. collating sequence) of tables in the source file
- the size of the generated binary file.

`–o` *output*

output must be a file (option –VSINIX) or a directory (default or option –V5.4). With option –V5.4 or by default, ic writes the generated files and the shell script to the directory *output*. If the –M option is set, @*modifier* is appended to the directory name. It is your responsibility to ensure that the name does not exceed a maximum length of 14 characters.

When the –VSINIX option is set, ic writes the generated binary database to the file *output*.

–o *output* not specified:

The name of the output file or output directory is the name of the CODESET section in the source file.

source

Name of the source file to be compiled.

source not specified: ic reads the source text from standard input.

EXIT STATUS

0 for successful compilation

≠0 if a binary file or the intermediate files could not be generated due to errors

ERROR MESSAGES

Four types of error message are issued by the ic compiler:

[warning]

The compiler has detected some kind of problem which has a high probability of being an error but may also be what the user wanted and has no adverse effect on the correctness of the generated file(s).

[error *nn*]

The compiler has detected an error severe enough to inhibit the generation of a correct binary file or of correct intermediate files.

[fatal error]

The compiler has detected an error which makes it impossible to proceed with the compilation. This error usually occurs during compilation of the code table.

[fatal bug]

An error message of this kind is issued when internal errors occur in the compiler. As a side effect, temporary files which may be needed to fix the bug are stored in the /tmp directory.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

FILES

/tmp/ic*XXXXXX*

Temporary files

LC_TIME

Intermediate file created by ic –V5.4

colltbl

Intermediate file created by ic –V5.4

chrtbl

Intermediate file created by ic –V5.4

montbl

Intermediate file created by ic –V5.4

Xopen_info

Intermediate file created by ic –V5.4

`install_locale`
: Shell script created by `ic -V5.4`.

  This shell script may only be run by the system administrator.

SEE ALSO

`what`(1).

Programmer's Guide, *Internationalization – Localization*.

User's Reference Manual

iconv(1)

NAME

`iconv -` international codeset conversion

SYNOPSIS

`iconv –f` *fromcode* `–t` *tocode* [`––`] [*file ...*]

DESCRIPTION

`iconv` reads input characters from a file or from standard input, converts the encoding of these characters and writes the results to standard output.

The conversions that can be performed with `iconv` are defined in conversion tables located in the `/usr/lib/iconv` directory. Only the available conversion tables can be used.

`iconv` is typically used to convert characters from the ISO 8859-1 character set to an ISO 646 ASCII variant codeset for a particular language and vice versa (see **EXAMPLES**).

OPTIONS

`–f` *fromcode*
`–t` *tocode*

`iconv` expects the conversion table to be in the file `/usr/lib/iconv/`*fromcode*`.`*tocode*`.t`. Any character that does not exist in the target codeset is converted to an underscore "_".

`––` If *file* begins with a dash (–), the end of the command line options must be marked with `––`.

file Name of the file for which codeset conversion is to be performed. It is possible to specify several files, which are then processed in one after the other.

file not specified: `iconv` reads from standard input.

ERROR MESSAGES

`Not supported` *xx* `to` *yy*

`iconv` does not support the requested conversion from codeset *xx* to target codeset *yy*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

List all conversion tables:

```
$ ls /usr/lib/iconv
646da.8859.t  646fr.8859.t  8859.646da.t  8859.646fr.t
646de.8859.t  646it.8859.t  8859.646de.t  8859.646it.t
646en.8859.t  646sv.8859.t  8859.646en.t  8859.646sv.t
646es.8859.t  8859.646.t    8859.646es.t  iconv_data
```

Example 2

Convert the file named `letter` and save the result in a file named `letter.conv`. The conversion is to be made from the German variant of the ISO 646 codeset (ASCII derivative) to the target codeset ISO 8859-1:

```
$ iconv –f 646de –t 8859 letter > letter.conv
```

FILES

`/usr/lib/iconv`
: Directory containing the standard conversion tables for codeset conversion

`/usr/lib/iconv/iconv_data`
: Auxiliary file for `iconv`

`/usr/lib/iconv/*.t`
: Conversion tables

SEE ALSO

`gencat`(1), `iconv`(3C), `iconv`(5), `iconv_tab`(5).

User's Reference Manual

id(1)

NAME

`id -` print user and group IDs and names

SYNOPSIS

`id` [*option ...*] [*username*]

DESCRIPTION

`id` writes the following to standard output for the invoking process:

- the user ID (UID)
- the login name
- the group ID (GID)
- the group name

If the effective and real IDs/names are not identical, both are printed.

Interaction with su

`id` reports information on the **current** login name. Thus, if you enter `id` after using the `su` command to switch login names (i.e. become another user), `id` will print information on your **new** name.

The `who` and `who am i` commands, by contrast, always return information on the name under which you originally logged in.

OPTIONS

The `–G`, `–g`, and `–u` options cannot be combined.

No option specified
: In addition to the ID and login name of the user, `id` reports all the groups to which the invoking process belongs and all the groups to which the invoking user belongs.

  This is equivalent to the old `–a` option, which will continue to be supported.

`–G` `id` outputs all group numbers.

`–g` `id` outputs only the effective group numbers.

`–n` The corresponding names are output instead of numbers.

`–r` The real ID is output instead of the effective ID. `–r` cannot be combined with `–G`.

`–u` `id` only outputs the effective user numbers.

`––` If *username* begins with a dash (–), the end of the command line options must be marked with ––.

username
: `id` outputs information on the specified user.

EXAMPLES

To check your current user ID, group ID, and their corresponding names, you enter:

```
$ id
```

`id` might then report the following:

```
uid=115(phil)            gid=1(other)
```

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

FILES

`/etc/passwd`
: Password file containing login names and, among other things, the associated user and group IDs.

`/etc/group`
: Group file containing group names and the associated group IDs and login names.

SEE ALSO

`logname`(1), `newgrp`(1), `su`(1), `who`(1), `getuid`(2).

User's Reference Manual

iecho(1)

NAME

`iecho -` echo message from message catalog

SYNOPSIS

`iecho` [`–n`] [[*catalog*] *set-number*] *message-number default-text*

DESCRIPTION

`iecho` writes a message to standard output, by default appending a new-line character.

OPTIONS

`–n` By default a new-line character is appended to the message text. The `–n` option suppresses this new-line character.

catalog
The message catalog to be searched. If *catalog* is not specified, `iecho` tries to use the value of the `NL_CATD` environment variable. If `NL_CATD` is not set, `iecho` reports an error and exits.

set-number
If *set-number* is not specified, `iecho` tries to use the value of the `NL_SETNR` environment variable. If `NL_SETNR` is not set, `iecho` reports an error and exits.

message-number
`iecho` tries to read the message with the specified *message-number* in the specified set of the specified message catalog. If it finds the message, it echos it to standard output. If not, it echos the error message *default-text* and exits.

default-text
If `iecho` fails to find the message, it echos the error message *default-text* and exits. `iecho` can only process one text argument at a time.

EXIT STATUS

`0` on success

`1` on error

ENVIRONMENT VARIABLES

`NL_CATD`
Defines the name of the message catalog

`NL_SETNR`
Defines the number of the message set

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

You need to internationalize the following output:

```
$ echo Hi $USER, the latest recipes are in $PATH !
```

Proceed as follows:

```
$ NL_CATD=foo; NL_SETNR=2; export NL_CATD NL_SETNR
```

```
$ iecho -n l "Hi "
$ echo -n $USER
$ iecho -n 2 ", the latest recipes are in "
$ echo $PATH !
```

SEE ALSO

iput(1).

Programmer's Guide, *Internationalization – Localization*.

User's Reference Manual

indxbib(1)
BSD Compatibility Package

NAME

`indxbib` - create an inverted index to a bibliographic database

SYNOPSIS

`/usr/ucb/indxbib` *database-file* ...

DESCRIPTION

`indxbib` makes an inverted index to the named *database-file* (which must reside within the current directory), typically for use by `lookbib` and `refer`. A *database* contains bibliographic references (or other kinds of information) separated by blank lines.

A bibliographic reference is a set of lines, constituting fields of bibliographic information. Each field starts on a line beginning with a "`%`", followed by a key-letter, then a blank, and finally the contents of the field, which may continue until the next line starting with "`%`" (see `addbib`).

`indxbib` is a shell script that calls two programs: `mkey` and `inv`. `mkey` truncates words to 6 characters, and maps upper case to lower case. It also discards words shorter than 3 characters, words among the 100 most common English words, and numbers (dates) < 1900 or > 2000. These parameters can be changed.

`indxbib` creates an entry file (with a `.ia` suffix), a posting file (`.ib`), and a tag file (`.ic`), in the working directory.

NOTES

All dates should probably be indexed, since many disciplines refer to literature written in the 1800s or earlier.

`indxbib` does not recognize pathnames.

FILES

`/usr/ucblib/reftools/mkey`
`/usr/ucblib/reftools/inv`
`*.ia`
entry file
`*.ib` posting file
`*.ic` tag file
`*.ig` reference file

SEE ALSO

`addbib`(1), `lookbib`(1), `refer`(1), `roffbib`(1), `sortbib`(1).

User's Reference Manual

ipcrm(1)

NAME

`ipcrm` - remove interprocess communication facilities

SYNOPSIS

`ipcrm` [*option* ...]

DESCRIPTION

The command `ipcrm` can be used to remove one or more semaphores, message queues, shared memory segments or interprocess communication facilities. These can be specified either by their identifiers or by the key with which each such IPC facility was created.

The identifiers and keys of IPC facilities can be listed with the help of the `ipcs` command (`ID` and `KEY` columns).

You will find detailed information on the removal of a message queue, a shared memory segment, or a semaphore in the *Programmer's Reference Manual* under the descriptions of `msgctl(2)`, `shmctl(2)` and `semctl(2)`, respectively.

OPTIONS

–`q` *msgqid*
: Removes the message queue identified as *msgqid* from the system and destroys the data structures associated with it.

msgqid
: Identifier of the message queue to be removed. This identifier is displayed by the `ipcs` command in the `ID` column.

–`Q` *msgkey*
: Removes the message queue identifier *msgkey* from the system and destroys the data structures associated with it.

msgkey
: Key of the message queue to be removed. This key is displayed by the `ipcs` command in the `KEY` column.

–`s` *semid*
: Removes the semaphore identifier *semid* from the system and destroys the data structures associated with it.

semid
: Identifier of the semaphore set to be removed. This identifier is displayed by the `ipcs` command in the `KID` column.

–`S` *semkey*
: Removes the semaphore set identified as *semkey* from the system and destroys the data structures associated with it.

semkey
: Key of the semaphore set to be removed. This key is displayed by the `ipcs` command in the `KEY` column.

–`m` *shmid*
: Removes the shared memory identifier *shmid* from the system and destroys the shared memory segment and data structures associated with it.

shmid
: Identifier of the shared memory segment to be removed. This identifier is displayed by the `ipcs`

command in the `ID` column.

–`M` *shmkey*
Removes the shared memory segment identified by *shmkey* from the system and destroys the data structures associated with it.

shmkey
Key of the shared memory segment to be removed. This key is displayed by the `ipcs` command in the `KEY` column.

EXAMPLES

Using `ipcs` to produce a report on the status of the interprocess communication facilities and then removing the message queue with the identifier 40 from the system:

```
$ ipcs
Time is 248bc52b
IPC status from /dev/kmem as of Tue Jun  6 13:35:55 1989
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
q     40 0x0000004b -Rrw-rw-rw-  michael    qm234
Semaphores:
$ ipcrm -q 40
```

SEE ALSO

`ipcs`(1), `msgctl`(2), `msgget`(2), `msgop`(2), `semctl`(2), `semget`(2), `semop`(2), `shmctl`(2), `shmget`(2), `shmop`(2).

User's Reference Manual

ipcs(1)

NAME

`ipcs` - report interprocess communication facilities status

SYNOPSIS

`ipcs` [*option* ...]

DESCRIPTION

`ipcs` prints information about active interprocess communication facilities (IPC facilities). A number of options can be used to control

- the type of IPC facility for which information is output
- the kind of information provided.

Since the status of IPC facilities can change while `ipcs` is running, the information returned by `ipcs` is guaranteed to be accurate only at the time it was retrieved.

OPTIONS

No option specified:

Prints information in short format for

- message queues
- shared memory segments
- semaphores

that are currently active in the system. The meanings of individual output columns are described under the **OUTPUT** section.

There are options for defining the type of IPC facility and options for defining the type of information.

Defining the type of IPC facility

`-q` Prints information about active message queues.

`-m` Prints information about active shared memory segments.

`-s` Prints information about active semaphores.

If the `-q`, `-m` or `-s` option is set, only information about the corresponding IPC facility is printed. The options can also be combined. If none of the options are set, information will be provided on all types of IPC facilities.

Defining the type of information

The following section describes the options that select the type of information to be provided for the facilities specified by the `-q`, `-m`, and `-s` options. See the **OUTPUT** section for the meanings of individual output columns.

`-a` Sets all options that define the type of information to display. This is a shorthand notation for `-b`, `-c`, `-o`, `-p`, and `-t`.

`-b` Prints the biggest allowable size for each respective IPC facility.

For message queues: the maximum number of bytes in a message to be placed on the queue.

For shared memory: the size of the memory segments.

For semaphores: the number of semaphores in each set of semaphores.

`-c` Prints the login name and group name of the creator of the IPC facility.

`-o` Prints the following information on outstanding IPC facilities:

- the number of messages on queue,

- the total number of bytes in messages on queue, and
- the number of processes attached to shared memory segments.

`-p` Prints the following process ID information:

- process ID of the last process that sent a message,
- process ID of the last process that received a message,
- process ID of the process that created a shared memory segment, and
- process ID of the last process to attach or detach on shared memory segments.

`-t` Prints time information. The time of the last control operation that changed the access permissions for all IPC facilities is displayed.

For message queues: time when the system call `msgrcv(2)` (receive message from queue) or `msgsnd(2)` (send message to queue) was last used.

For shared memory: time when the system call `shmat(2)` or `shmdt(2)` was last used.

For semaphores: time when the system call `semop(2)` was last used on a semaphore.

Additional arguments

`-C` *corefile*
Uses the file named *corefile* instead of `/dev/kmem`. This option can only be used by `root` or by users with permission to access `/dev/kmem`.

`-N` *namelist*
Uses the file named *namelist* instead of `/stand/unix`. This option can only be used by `root` or by users with permission to access `/dev/kmem`.

`-X` Prints information about XENIX interprocess communication, in addition to the standard IPC facilities status. The XENIX IPC information describes a second set of semaphores and shared memory.

The `-p` option does not print process ID information for XENIX shared memory. The `-t` option does not print time information about XENIX semaphores and shared memory.

If you specify either the `-C` or `-N` option, the real and effective user ID and group ID are set to the real user ID and group ID of the user invoking `ipcs`.

OUTPUT

The column headings and the meaning of the columns in an `ipcs` listing are given below; the letters in parentheses (...) indicate the options that cause the corresponding heading to appear; "all" means that the heading always appears.

`CBYTES` (`a`, `o`)
The number of bytes in messages currently outstanding on the associated message queue.

`CGROUP` (`a`, `c`)
The group name to which the creator of the IPC facility belongs.

`CREATOR` (`a`, `c`)
The login name of the creator of the IPC facility entry.

`CTIME` (`a`, `t`)
The time when the associated entry was created or changed.

`GROUP` (all)
The group name of the group to which the owner of the IPC facility belongs.

`ID` (all)
The identifier for the IPC facility.

`KEY` (all)
The key used as an argument when creating the IPC facility with `msgget(2)` (create message queue) or `semget(2)` (create semaphore set).

LRPID (a, p)
The process ID of the last process to receive a message from the associated queue.

LSPID (a, p)
The process ID of the last process to send a message to the associated queue.

MODE (all)
The facility access modes and status indicators. The mode consists of 11 characters that are interpreted as follows:

The first character can be:

S if a process is waiting on a msgsnd(2) .

D if the associated shared memory segment has been removed. It disappears when the last process attached to the segment detaches it.

– if no process is waiting on a msgsnd(2) .

The second character can be:

R if a process is waiting on a msgrcv(2) .

C if the associated shared memory segment is to be cleared when the first attach is executed.

– if no process is waiting on a msgrcv(2) .

The next 9 characters are interpreted as three sets of three bits each:

The first set refers to the access permissions of the owner of the IPC facility.

The next set represents the access permissions of others belonging to the same user group as the owner of the facility entry.

The third set refers to the access permissions of all other users.

Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused.

The permissions are specified as follows:

r if read permission is granted

w if write permission is granted

a if permission to alter the entry is granted

– if the indicated permission is not granted.

NSEMS (a, b)
The number of semaphores in the associated semaphore set.

OTIME (a, t)
The time the last semaphore operation was completed on the associated semaphore set.

OWNER (all)
The login name of the owner of the IPC facility entry.

QBYTES (a, b)
The maximum number of bytes allowed in messages outstanding on the associated message queue.

QNUM (a, o)
The number of messages currently outstanding on the associated message queue.

RTIME (a, t)
The time the last message was received from the associated message queue.

STIME (a, t)
The time the last message was sent to the associated message queue.

NATTCH (a, o)
The number of processes attached to the associated shared memory segment.

SEGSZ (a, b)
The size of the associated shared memory segment.

`CPID` (`a`, `p`)
: The process ID of the creator of the shared memory segment.

`LPID` (`a`, `p`)
: The process ID of the last process to attach or detach the shared memory segment.

`ATIME` (`a`, `t`)
: The time the last attach was completed to the associated shared memory segment.

`DTIME` (`a`, `t`)
: The time the last detach was completed on the associated shared memory segment.

`T` (all)
: Type of the IPC facility:

  `q` for message queue

  `m` for shared memory

  `s` for semaphore.

EXAMPLES

The `server` program sets up a message queue. You first start this program in the background and then check the status of the interprocess communication facilities with `ipcs`:

```
$ server &
$ ipcs
IPC status from /dev/kmem as of Tue Nov 17 10:12:23 MET 1998
T      ID      KEY         MODE       OWNER    GROUP
Message Queues:
q      700 0x00000020 --rw--w-r--     root    vsxg0
Shared Memory:
Semaphores:
s        0 0x2520509a --ra-ra-ra-     root     root
```

FILES

`/stand/unix`
: System name list

`/dev/kmem`
: File containing a virtual memory map of the operating system kernel

`/etc/passwd`
: The `/etc/passwd` file contains all the login names that have been set up

`/etc/group`
: The `/etc/group` file contains all the group names that have been set up

SEE ALSO

`ipcrm`(1), `msgctl`(2), `msgget`(2), `msgop`(2), `semctl`(2), `semget`(2), `semop`(2), `shmctl`(2), `shmop`(2).

User's Reference Manual

iput(1)

NAME

`iput` - query international database

SYNOPSIS

`iput` *string* ...

DESCRIPTION

`iput` provides access to an international database in the same manner that `tput` provides access to the terminfo database.

`iput` takes arguments in the form of one or more constants defined in the current international database. `iput` searches this database for the specified constants and writes their values to standard output.

The database that `iput` accesses is governed by the value of the NLS environment variables `LANG`, `LC_TIME`, `LC_NUMERIC` and `LC_MONETARY` (see **ENVIRONMENT VARIABLES**).

OPERANDS

string

Names of the constants to be inspected.

You can specify any of the constants defined in the `<langinfo.h>` header file.

ENVIRONMENT VARIABLES

The NLS environment variables `LANG`, `LC_TIME`, `LC_NUMERIC` and `LC_MONETARY` determine which international database `iput` will access. The particular variable which governs the database selected when you call `iput` depends on the categories to which the specified constants belong [see `langinfo(5)`]. If any of the variables `LC_TIME`, `LC_NUMERIC` or `LC_MONETARY` is undefined or assigned the null string, it defaults to the value of `LANG`.

Example:

```
$ iput DAY_1 ABDAY_1
```

The constants `DAY_1` and `ABDAY_1` both belong to the `LC_TIME` category, so the value of the `LC_TIME` variable determines which database `iput` will access. If the `LC_TIME` variable is undefined or null, `iput` will access the database associated with the value of the `LANG` variable.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

```
$ LANG=De_DE.88591 iput YESSTR
```

displays the "yes string" applicable in the German locale `LANG=De_DE.88591`. (The "yes string" is the string you use to answer yes to questions from the system.)

```
$ iput DAY_1 ABDAY_1
```

displays the name of the first day of the week in full nd abbreviated forms [see `date(1)`].

EXIT STATUS

`0` on success

`1` for an `iput` usage error

2 if the database cannot be found or is defective

3 if the constant you specified is not defined in the database

FILES

`/usr/lib/nls/intlinfo`
directory for international databases

SEE ALSO

`setlocale`(3C).

Programmer's Guide, *Internationalization – Localization*.

User's Reference Manual

join(1)

NAME

`join -` join two files on identical-valued field

SYNOPSIS

`join` [*option ...*] *file1 file2*

DESCRIPTION

`join` compares two files on the basis of relations ("join fields") and joins all pairs of lines with identical join fields. The result is displayed on standard output.

When `join` is invoked, a join field on which the files are to be compared must be specified for each of the two files. Each field is bounded by a pair of field separators. `join` compares each line in the first file with lines in the second and displays one output line on the standard output for each pair of lines with identical join fields. The output line comprises specific fields from both lines.

Before the call

Each input file must be sorted so that the join fields are arranged in the currently valid collating sequence [see `sort`(1)]. If the default field separator is used (`join` without the `–t` option), leading separators must be ignored (see `sort`, option `–b`) when the files are sorted. However, if you invoke `join` with option `–t`, leading field separators must be taken into account when sorting the files (see `sort` without option `–b`).

OPTIONS

No option specified:

The first field in a line is the default join field for both files; the default separators are blanks, tabs, and newline characters. Multiple field separators count as one field separator, and leading separators are ignored.

`join` displays one output line on the standard output for each pair of lines with identical join fields. Each output line consists of the following entries in the given order:

- the common field
- the rest of the line from the first file
- the rest of the line from the second file

The default output field separator is a blank.

`–1` *m* or `–2` *m*

The *m*th field is defined as the join field for the *n*th file. *n* stands for 1 or 2. Enter an integer greater than or equal to 1 for *m*.

`–1` or `–2` not specified: The join field for the *n*th file is the 1st field.

The options `–1` and `–2` correspond to the old option `–j`, which will continue to be supported, but must not be combined with the new synopsis. The following correspondence exists between the old and the new synopsis:

`–j` *m* is equivalent to `–1` *m* `–2` *m*

`–j1` *m* is equivalent to `–1` *m*

`–j2` *m* is equivalent to `–2` *m*

`–a` *n* In addition to the normal output, prints each line of file *n* for which no matching join field can be found in the other file.

n can be 1 or 2. To generate output for both files you can enter `–a 1 –a 2`.

`–a` cannot be combined with `–v`.

`–e` *string*

Replaces empty output fields with the specified string.

–o *list*

`join` changes the output line format, so that each output line comprises the individual fields specified in *list*. The common field is not printed unless you explicitly specify it in *list*.

The *list* you specify must consist of elements in the form *n*.*m*, where *n* is either 1 or 2, and *m* is greater than or equal to 1. Each element in the form *n*.*m* stands for the *m*th field in the *n*th file. Enter 0 (zero) for the comparison field. The elements are separated by commas.

Blanks or tabs may still be used as separators, but must not be combined with the new syntax.

–t *c* Defines character *c* as a field separator for both input and output lines. Each occurrence of *c* is interpreted as a field separator, i.e.

- two consecutive *c* separators designate an empty field, and
- a leading *c* is significant and designates an empty first field.

In addition, the newline character acts as a field separator for the input lines.

The default field separators (blanks and tabs) are interpreted as field separators only if you specify them as a value for *c*.

–v (v – vice versa) `join` only outputs the lines of the *n*th input file whose comparison field does not match the comparison field of the other file.

You can specify 1 or 2 for *n*. If the output for both files is produced, specify `–v 1 –v 2`.

`–v` cannot be combined with `–a`.

–– If *file1* begins with a dash (–), the end of the command-line options must be marked with ––.

file1 file2

Names of the two files to be joined on the basis of common fields by `join`.

If you use a dash (–) as the name for *file1*, `join` reads from standard input.

Caution:

If the files are not sorted on their join fields, `join` will not process all lines!

Problems may arise if a numeric file name (e.g. 1.2) is specified for *file1* and the `–o` option is used immediately before this file name is listed. To avoid such conflicts, a numeric file name should be preceded by a slash (e.g. ./1.2).

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_COLLATE` governs the collating sequence.

`LC_TYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES`, `LC_COLLATE` or `LC_TYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

In the file `place`, a place is assigned to a name. In the file `amount`, an amount and a date are assigned to the same names. Both files are sorted by name. `join` is to join the two files on the names:

Contents of `place`:

```
Albert Buffalo
Hugh Washington
Irene Philadelphia
```

Contents of `amount`:

```
Albert   287.56  20.03.88
Hugh      23.15  25.06.87
Hugh     167.87  16.12.87
Irene   1212.12  12.12.88
Irene      1.98  01.01.88
```

Join the two files on the first join field:

```
$ join place amount
Albert Buffalo 287.56 20.03.88
Hugh Washington 23.15 25.06.87
Hugh Washington 167.87 16.12.87
Irene Philadelphia 1212.12 12.12.88
Irene Philadelphia 1.98 01.01.88
```

Join the two files and format in columns with `awk`:

```
$ join place amount | awk '{printf("%-10s %-15s %-10s %-10s\n", $1,$2,$3,$4)}'
Albert    Buffalo          287.56    20.03.88
Hugh      Washington        23.15    25.06.87
Hugh      Washington       167.87    16.12.87
Irene     Philadelphia    1212.12    12.12.88
Irene     Philadelphia       1.98    01.01.88
```

Example 2

In the file `city`, a name is assigned to a city. In the file `amount` (see Example 1), an amount and a date are assigned to a name. `city` is sorted by cities, `amount` by names. `join` is to join the two files on the names.

Contents of `city`:

```
Buffalo      Albert
Buffalo      Frank
Washington   Hugh
New York     Eric
Philadelphia Irene
```

In this example, the join field for `city` is field 2, while that of `amount` is field 1.

Before the files are joined, `city` must be sorted on field 2. The output is subsequently formatted into columns with `awk`:

```
$ sort -b -k 2 city | join -1 2 - amount | \
> awk '{printf("%-10s %-15s %-10s %-10s\n",$1,$2,$3,$4)}'
Albert     Buffalo         287.56     20.03.88
Hugh       Washington      23.15      25.06.87
Hugh       Washington      167.87     16.12.87
Irene      Philadelphia    1212.12    12.12.88
Irene      Philadelphia    1.98       01.01.88
```

SEE ALSO

`awk(1)`, `comm(1)`, `sort(1)`, `uniq(1)`.

User's Reference Manual

jsh(1)

NAME

`jsh` - Bourne shell with job control

SYNOPSIS

`jsh` [*option* ...] [*argument* ...]

DESCRIPTION

`jsh` is a Bourne shell (`sh`) with job control enabled in addition to the normal functionality of the Bourne shell. This allows you to:

- stop jobs,
- move jobs to the background,
- bring jobs from the background to the foreground, and
- easily identify and access jobs via job numbers.

As a rule, job control can only be used in an interactive shell, i.e. a login shell or a subshell. Shells running shell scripts do not benefit from the added functionality of job control.

The enhancements provided by job control are the introduction of:

- identifiable jobs with various states,
- new shell built-ins, and
- extensions to existing shell built-in.

The possible entries for *option* and *argument* are discussed in detail in the description of the Bourne shell `sh`. There is no difference between the two shells in this respect.

MODE OF OPERATION

`jsh` treats every command or pipeline that the user enters at the terminal as a job. All jobs exists in one of the following three states:

`Foreground`
: A foreground job has read and write access to the controlling terminal.

`Background`
: A background job is denied read access and has conditional write access to the controlling terminal [see `stty(1)`].

`Stopped`
: A stopped job is a job that has been suspended, usually as a result of a `SIGTSTP` signal [see `signal(5)`].

Every job started by `jsh` is assigned a positive integer called a job number, which is tracked by the shell and used as an identifier to indicate a specific job. In addition, 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:

`%`*jobid*

where *jobid* may be in any of the following formats:

`%` or `+`
: the current job

`–`
: the previous job

`?`*string*
: the job for which the command line uniquely contains *string*

num the job with job number *num*

prefix
the job whose command name begins with the string *prefix*. The *prefix* must not contain blanks unless they are quoted.

Example: The `ls –l` ... command running in the background could be referred to as `%ls`.

You can stop a job by sending a `STOP` signal to it with the key combination CTRL-Z. `jsh` will then indicate that the job has been stopped and will display a prompt. The CTRL-Z combination produces an immediate response (like the interrupt key DEL).

Caution:
If you are using `jsh` together with `shl`, you must redefine the `switch` character [see `stty(1)`] to prevent the `suspend` character from being deactivated.

COMMANDS

The following additional shell built-ins may be entered after the PS1 prompt (default: $):

`bg` [`%`*jobid* ...]
Resumes the execution of a stopped job in the background. *jobid* designates the stopped job(s). If *jobid* is omitted, it defaults to the current job.

`fg` [`%`*jobid* ...]
Resumes the execution of a stopped job in the foreground, or moves an executing background job into the foreground. *jobid* designates the job(s) to be processed. If *jobid* is omitted, it defaults to the current job.

`jobs` [`–l`|`–p`] [`%`*jobid* ...] **Format 1**

`jobs –x` *command* [*argument* ...] **Format 2**

Format 1: Print information about jobs

Prints information on standard output about 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 modify the output of `jobs`:

`–l`
Writes the process group ID and working directory of the jobs to standard output.

`–p` Writes only the process group ID of the jobs to standard output. .RE

Format 2: Apply command to jobs

This command replaces any *jobid* found in *command* or *argument* with the corresponding process group ID and then executes the resulting command with the resulting arguments.

`kill` [*signal*] `%`*jobid*
This is a Bourne shell built-in version of the `kill` command. It provides the same functionality as the `kill` command, but allows you to identify a job with a *jobid* instead of a process ID.

`stop %`*jobid* ...
Stops the execution of the background job(s) identified by *jobid*.

`suspend`
Stops the execution of the current `jsh` shell (but not if it is a login shell).

`wait` [`%`*jobid* ...]
This version of the Bourne shell built-in `wait` accepts *jobid* as an argument. If *jobid* is omitted, `wait` behaves like the normal `wait` built-in; otherwise, it waits for the background job(s) to terminate.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In file name generation patterns in square brackets, the `LC_COLLATE` environment variable governs the scope

of character ranges, equivalence classes and collating elements, and the LC_CTYPE environment variable governs the scope of character classes. LC_CTYPE also governs which characters in the current character set are defined as letters.

If LC_MESSAGES, LC_COLLATE or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

The following section contains a brief example demonstrating job handling in the jsh shell.

The first step is to call the shell by entering

```
$ jsh
```

To avoid confusion, it is advisable to modify the prompt:

```
$ PS1="jsh> "
```

Let us assume you now want to look for a file named test. To do this, you run the find command in the background:

```
jsh> find / –name test –print &
```

The shell then displays job ID 1 and process ID 28071:

```
[1] 28071
```

You now print a list of all current jobs:

```
jsh> jobs
[1] + Running                    find / –name test –print
```

Next, you stop job number 1 (i.e. the find command):

```
jsh> stop %1
```

You then again print list of all current jobs:

```
jsh> jobs
[1] + Stopped (signal)           find / –name test –print
```

and resume execution of the current job (i.e. the find command) in the background:

```
jsh> bg %%
```

The shell displays the resumed command:

```
[1] find / –name test –print &
```

You can now terminate the find job with:

```
jsh> kill %find
```

and print a fresh list of all current jobs:

```
jsh> jobs
[1] + Terminated                 find / –name test –print
```

Finally, you exit the jsh shell with:

```
jsh> exit
```

and return to the Bourne shell with its standard prompt:

```
$
```

SEE ALSO

sh(1) .

User's Reference Manual

kill(1)

NAME

`kill -` send signals to processes

SYNOPSIS

Format 1

`kill` **[** `-s` *signal* **]** *pid ...*

Format 2

`kill` **[** *–signalno* **]** *pid ...*

Format 3

`kill –s` *signal –– –pgid ...*

Format 4

`kill –l` **[** *number* **]**

DESCRIPTION

The `kill` command sends the specified signal to a set of processes indicated by a process number (PID). The `ps(1)` command can be used to find the process number of any process to which you want to send a signal.

OPTIONS

Format 1a: Send signals to processes (Korn shell or XPG4 shell only)

`kill` [`–s` *signal*] *pid ...*

`–s` *signal*

Signal to be sent to the processes. This signal can be given in the form of a number or a symbolic name. The symbolic signal name is the name as it appears in the header file `<sys/signal.h>`, without the `SIG` prefix. No distinction is made between uppercase and lowercase letters. A list of these names can be printed with the `–l` option.

Any signal defined in the header file `<sys/signal.h>` may be specified. The signals below are significant on command level:

Signal no.	Symbolic name	Meaning
1	SIGHUP	Halt if the user hangs up
2	SIGINT	Interrupt via <DEL>
3	SIGQUIT	Quit
9	SIGKILL	Kill, terminate process unconditionally
15	SIGTERM	Software termination, terminate process gracefully

`–s` *signal* not specified:

`kill` sends the signal `SIGTERM`(15) to the specified processes. This will normally kill processes that do not catch or ignore the signal.

`–s` *signal* is equivalent to the old Bourne shell option *–signal* (cf. **Format 1b** and **EXAMPLES**), which will continue to be supported.

pid Number (ID) of the process to which a signal is to be sent.

Users without system administrator privileges may only send signals to their own processes.

The system administrator can send signals to all processes.

Current process IDs can be listed with the `ps` command.

A *pid* value of 0 has a special meaning: the signal is sent to all processes in your process group.

Format 1b: Send signals to processes

`kill` [*–signalno*] *pid ...*

See **Format 1a**, whereby *–signalno* behaves like `–s` *signal*.

Caution:

The external `/usr/bin/kill` command is the same as the `kill` command in the Bourne shell, i.□e. it only recognizes the syntax specified under Format 1b.

Format 2: Send signals to process groups

`kill –s` *signal –– –pgid ...*

`–s` *signal*
: Signal to be sent to the processes of a process group. This signal can be given in the form of a number or a symbolic name. The symbolic signal name is the name as it appears in the header file `<sys/signal.h>`, without the `SIG` prefix. A list of these names can be printed with the `–l` option (see also **Format 1a**).

`––` End of the list of options. Must be specified so that *–pgid* is not misinterpreted as an option.

–pgid The signal is sent to all processes with process group ID *pgid*.

Users without system administrator privileges may only send signals to their own process groups.

The system administrator can send signals to all process groups.

If you specify a value of `1` for *pgid*, the designated *signal* will be sent to every process whose real user ID matches the effective user ID of the `kill` command. As privileged user you send the signal to all processes, with the exception of a number of system processes.

Format 3: List symbolic signal names

`kill –l` [*number*]

`–l` A list of all symbolic signal names is printed.

number
: If *number* corresponds to the exit status of a process that was aborted by a signal, `kill` outputs the symbolic name of this signal. If this is not the case, and *number* corresponds to a signal number, `kill` outputs the symbolic name of the signal.

ERROR MESSAGES

`no such process`

You specified an invalid value for *pid*.

`no such process group`

You specified an invalid value for *pgid*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

Example 1a (Korn shell or XPG4 shell only)

The signal `SIGKILL`(9) is used to terminate process number 312:

`$ kill -s KILL 312` or `$ kill -s 9 312`

Example 1b

The signal SIGKILL(9) is used to terminate process number 312:

```
$ kill -9 312
```

Example 2

The find commands running in the background are terminated by sending the signal SIGTERM(15) to all processes with your process group ID. The shell (sh) associated with the same process group by default ignores SIGTERM.

```
$ ps
   PID  PGID   SID TTY      TIME COMD
 13320 13278 13278 tty004   0:08 find
 13324 13278 13278 tty004   0:00 ps
 13278 13278 13278 tty004   0:01 sh
 13322 13278 13278 tty004   0:04 find
$ kill -s TERM -- -13278

$ ps
   PID  PGID   SID TTY      TIME COMD
 13336 13278 13278 tty004   0:00 ps
 13278 13278 13278 tty004   0:01 sh
13320 Terminated
13322 Terminated
```

NOTES

kill exists both as an external command (/usr/bin/kill) and as a built-in shell command, both in the Korn shell ksh(1) as well as in the Bourne shell sh(1). The shell generates a new process to execute /usr/bin/kill.

FILES

<sys/signal.h>
: Header file in which the symbolic names of signals are defined.

SEE ALSO

ksh(1), ps(1), sh(1), trap(1), xpg4_sh(1), kill(2), signal(2), signal(5).

User's Reference Manual

ksh(1)

NAME

`ksh, rksh, alias, bg, break, continue, fc, fg, jobs, let, print, return, typeset, unalias, whence -` Korn shell

SYNOPSIS

`ksh` [*option* ...] [*file*] [*argument* ...]

DESCRIPTION

This command description is divided into the following sections:

- Major Korn shell features not available in the Bourne shell
- Call options
- Prompting
- Definitions
- Commands
- Reserved words
- Compound commands
- Comments
- Aliasing
- Tilde substitution
- Command substitution
- Korn shell variables and parameter substitution
- Blank interpretation
- File name generation
- Quoting metacharacters
- Arithmetic evaluation
- Conditional expressions
- File attributes
- String attributes and comparisons
- Algebraic comparisons between integers
- Negated and compound expressions
- I/O redirection
- Environment
- Functions
- Jobs
- Signals
- Execution
- Command re-entry
- In-line editing options
- `emacs` editing mode
- `vi` editing mode
- Input edit commands
- Motion edit commands

- Search edit commands
- Text modification edit commands
- Other edit commands
- Built-in commands

The Korn shell `ksh` is a programmable command interpreter that interprets and executes commands read from a terminal or a file.

`rksh` is a restricted version of the command interpreter `ksh`; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See **OPTIONS** below for the meaning of arguments to the shell.

Like the standard shell `sh(1)` and the C shell `csh(1)`, the Korn shell `ksh` is a user interface to the operating system kernel. The Korn shell is an extension of the standard Bourne shell and is essentially compatible with the Bourne shell in terms of syntax and semantics. Bourne shell scripts can therefore also be run in the Korn shell. The functionality of the C shell is as powerful as that of the Korn shell, but the C shell is more closely matched to the syntax of the C language. An overview of the enhanced functionality of the Korn shell is provided below for experienced users of the Bourne shell. Users who are not so familiar with the Bourne shell will find more detailed explanations in this manual under the general description of the Bourne shell `sh` and under the individual descriptions of the Bourne shell built-ins.

Major Korn shell features not available in the Bourne shell

- Two-way pipelines (| `&`) between commands
- Interactive selection from a word list with the `select` command
- The `time` statement
- Command name aliasing
- Tilde substitution for file names
- Extended command substitution syntax supporting arithmetic expressions
- The built-in `typeset` command can be used to set attributes for parameters to define their format, data type, etc.
- Support for single-dimensional arrays
- There are more automatically set parameters and more variables in the Korn shell.
- The file name generation mechanism also supports pattern lists. Patterns can be used to generate a single file name in I/O redirections.
- The built-in `let` command is available for integer arithmetic.
- Conditional expressions with similar functionality to the `test` command are provided as built-in statements.
- Functions can have local variables, are "exportable", and can be located via a search path.
- Job control can be activated with the `monitor` option of the built-in `set` command. You can thus access processes as easily as with the `jsh(1)` shell (Bourne shell with job control).
- When the command line is processed, built-in commands are located before functions.
- Positional parameters are saved before function calls and then restored.
- Command re-entry: commands issued to an interactive Korn shell are saved in a `history` file, which can be accessed using the built-in `fc` command. Simple text substitution can be performed on the command line before it is executed. There is also an option for editing command lines (in `emacs` or `vi` mode) from the history file before executing them.
- When the shell is invoked as a login shell, the file named by the value of the environment variable `ENV` can be called after `$HOME/.profile`.
- Some of the built-in commands have extra functionality; others are replaced by new commands or are not implemented.

Enhanced commands

Command	Extra features
`.`	Arguments
`:`	Arguments
`cd`	Substitution
`export`	Value assignment
`getopts`	Shell variables
`read`	Two-way pipelines
`readonly`	Value assignment
`set`	Array assignments, etc.
`shift`	Arithmetic expressions
`trap`	Special signals
`umask`	Symbolic arguments like `chmod`
`unset`	No readonly variables
`wait`	Job control

New commands

Command	Function
`alias`	Command aliasing
`bg`	Job control
`fc`	Access to history file
`fg`	Job control
`jobs`	Job control
`kill`	Job control
`let`	Integer arithmetic
`print`	Output mechanism like `echo`
`typeset`	Attributes for shell variables
`unalias`	Removing aliases
`whence`	Command classification

OPTIONS

Commands are read from `/etc/profile` and then from either `.profile` in the current directory or `$HOME/.profile`, if either file exists. Next, commands are read from the file whose name is generated by parameter substitution on the value of the shell variable `ENV`.

The following options are interpreted by `ksh` when it is invoked:

`-c` *command_string*
: Commands are read from *command_string*.

`-s`
: If the `-s` option is present or if *file* and *argument* are omitted, the Korn shell reads commands from standard input and writes its output to standard error, and built-in `ksh` commands write to standard output.

`-i` If the `-i` option is present or if standard input and output are associated with a terminal, an interactive Korn shell is called. In this case `TERM` is ignored (so that `kill 0` does not kill an interactive shell) and `INTR` is caught and ignored (so that `wait` is interruptible). In all cases, `QUIT` is ignored by the shell.

`-r` The `-r` option is used to invoke a restricted Korn shell.

The remaining options are described under the Korn shell built-in `set`.

`--` If *file* begins with a minus sign (-), the end of the command line options must be marked with `--`.

Note: In earlier versions of the shell, a single minus sign had the same effect that a double minus sign now has, namely to mark the end of the list of options. While a single minus sign can still be specified today, it does not have a recognizable effect.

file If the `-s` option is not present but *file* is, then a path search is performed to locate the script *file* to be executed. The script must have read permission. If the set-user-ID or set-group-ID bit has been set for the script, it is ignored. Commands are then read as described below.

argument

You can specify commands as arguments; the command name is passed as argument zero. Commands are described further below.

If the Korn shell is invoked by the system call `exec`, and the first character of argument zero is a minus sign (-), then the shell is assumed to be a `login` shell.

rksh only

`rksh` is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of `rksh` are identical to those of `sh`, except that the following are disallowed:

- changing directory [see `cd(1)`],
- setting the value of `SHELL`, `ENV`, or `PATH`,
- specifying path or command names containing /,
- redirecting output (`>`, `>|`, `<>`, and `>>`).

The restrictions above are enforced after `.profile` and the `ENV` files are interpreted.

When a command to be executed is found to be a shell procedure, `rksh` invokes `ksh` 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` 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 (i.e., `/usr/rbin`) that can be safely invoked by `rksh`. Some systems also provide a restricted editor `red(1)` .

GENERAL

Prompting

When the Korn shell is used interactively, it prompts with the expanded value of the `PS1` variable before reading a command.

If at any time a newline is typed and further input is needed to complete a command, then the secondary prompt is issued. The secondary prompt is generally the value of the `PS2` variable.

Definitions

A metacharacter is one of the following characters:

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

A `blank` is a tab or a space. An identifier is a sequence of letters, digits or underscores, starting with a letter

or underscore. Identifiers are used as names for functions and variables. A word is a sequence of characters separated by one or more non-quoted metacharacters.

Commands

A command is a sequence of characters in the programming language syntax of the Korn shell. The shell reads each command and carries out the desired action either directly or by invoking separate utilities.

A built-in (or special) command is a command that the Korn shell executes without creating a separate process. Except for the side effects documented below, most built-in commands can be implemented as separate utilities.

A simple command is a sequence of blank-separated words which may be preceded by a variable assignment list (see **Environment** below). 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. The value of a simple command is its exit status if it terminates normally, or (octal) `200+`*status* if it terminates abnormally [see `signal(5)` for a list of *status* values].

A pipeline is a sequence of one or more commands separated by the vertical bar character | (pipe symbol). The standard output of each command but the last is connected by a `pipe` system call to the standard input of the next command. Each command is run as a separate process; the Korn shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A command list is a sequence of one or more pipelines separated by the symbols

`;`, `&`, `&&`, or `||`

and optionally terminated by □
`;`, `&`, or `|&`

The 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 `|&` after a command list causes asynchronous execution of the preceding command or pipeline with a two-way pipe established to the parent shell.

The `-p` option of the `print` and `read` built-ins (described below) allows the parent shell to write to the standard input of the spawned command and read from its standard input respectively. The symbols `&&` and `||` cause the subsequent command list to be executed only if the preceding pipeline returns a zero or non-zero value respectively. An arbitrary number of newlines may appear in place of a semicolon as a delimiter in a command list.

A command is either a simple command or one of the following compound commands. Unless otherwise stated, the value returned by a command is always that of the last simple command executed in it.

Reserved words

The following reserved words are only recognized as the first word of a command provided they are not quoted.

```
if then elif else fi case esac for□
□
□
select while until do done { } function□
□
□
time [[ ]]
```

Words are only recognized if they are enclosed in delimiter characters (blanks, punctuation marks etc.). Therefore, in the case of { *command_list*`;` } for example, a blank must be entered after { and a semicolon must be placed before }.

Compound commands

`for` *identifier* [`in` *word_list*];`do` *command_list;*`done`

or

`for` *identifier* [`in` *word_list*] □
`do` *command_list* □
`done`

The `for` command can be used to repeatedly execute *command_list*. Each time a `for` loop is executed, *identifier* is set to the next word taken from the *word_list*. If *word_list* is omitted, the *command_list* is executed once for each positional parameter (`"$@"`) that is set. Execution ends when there are no more words in the *word_list*.

`select` *identifier* [`in` *word_list*] □
`do` *command_list* □
`done`

The `select` command can be used to repeatedly execute a given *command_list* under input control. `select` prints the set of words in *word_list* on standard output, each preceded by a number. The `PS3` prompt is then printed, and a line is read from the standard input. The content of the line read from standard input is saved in the variable `REPLY`. If this line matches the number of one of the listed words, then the value of the parameter *identifier* is set to the word corresponding to this number. If the line is empty, the *word_list* is printed again, and *identifier* is set to the name of the shell script. If `in` *word_list* is omitted, then the positional parameters are used instead (as is done with `for` loops). The `select` loop is executed repeatedly until a `break` built-in or end-of-file is encountered.

Example:

The following Korn shell script can be used to selectively print information on each individual file in the current directory:

```
select file in `ls`□
do□
  if [ -z "$file" ]□
  then□
     echo "Select number"□
  else□
     file $file□
     ls -lsid $file□
  fi□
done
```

`case` *word* `in` [[`(`]*pattern*[`|` *pattern*] ...`)` *command_list* `;;`] ... □
`esac`

The `case` command provides for conditional branching to a command list on the basis of a pattern. `case` executes the *command_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** below).

Caution:

The optional left parenthesis preceding *pattern* must be specified if `case` commands are used within command substitutions with `$`(...). This produces paired parentheses constructs in `$`(...).

`if` *command_list1* □
`then` *command_list2* [`elif` *command_list3* □
`then` *command_list4*] ... □
[`else` *command_list5*] □
`fi`

The `if` command can be used to conditionally execute a number of different command lists. The first list following `if` (*command_list1*) is executed as the condition, and if it returns a zero exit status (true), the list following the first `then` is executed (*command_list2*). Otherwise, the list following `elif` (*command_list3*) is executed as the next condition, and if its value is zero, the list following the next `then` (*command_list4*) is executed. Failing that, the `else` list (*command_list5*) is executed. If no `else` list or `then` list is executed, the `if` command returns a zero exit status.

`while` *command_list1* □
`do` *command_list2* □
`doneuntil` *command_list1* □
`do` *command_list2* □
`done`

The `while` and `until` commands define loops with exit conditions. A `while` command repeatedly executes the condition *command_list1*, and if the exit status of the last command in the list is zero (i.e. true), it executes the `do` list in the body of the loop (*command_list2*); otherwise the loop terminates. If no commands in *command_list2* (the `do` list) are executed, the `while` command returns a zero exit status.

`until` may be used in place of `while` to negate the loop termination test. The `until` command tests whether the exit condition is false (non-zero) and terminates the loop as soon as it returns an exit status of zero (true).

`(`*command_list*`)`

Executes *command_list* in a separate environment.

Caution:
If two adjacent left parentheses are needed for nesting, a space must be inserted between them to avoid arithmetic evaluation (see **Command substitution** below).

`{` *command_list* `;` `}`

command_list is simply executed in the current Korn shell.

Caution:
Note that unlike the parentheses `(` and `)`, the braces `{` and `}` are reserved words and must be typed at the beginning of a line or after a semicolon (`;`) in order to be recognized as such.

`[[` *expression* `]]`

Evaluates the conditional *expression* and returns a zero exit status if *expression* is true; otherwise 1.

See **Conditional expressions** below for a description of *expression*.

`function` *identifier* `{` *command_list* `;` `}` *identifier* `()` `{` *command_list* `;` `}`

Defines a function named *identifier*. This name is used to call the function like a command. The body of the function consists of the *command_list* that is enclosed within curly braces `{` and `}`.

Caution:
The left brace `{` must be followed by a space.

`time` *pipeline*

The commands in *pipeline* are executed and the elapsed time and the user and system time are reported on standard error.

SUBSTITUTIONS AND VARIABLES

Comments

A word beginning with the hash character `#` causes that word and all the following characters in the line to be ignored.

Aliasing

If the first word of a command is a defined alias, it is replaced by the text of the alias. An alias name consists of any number of characters excluding metacharacters, quoting characters, file expansion characters, parameter and command substitution characters, and the equal sign `=`. The replacement string can contain any valid Korn shell script including the metacharacters listed above. The first word of each command in the replaced text will in turn be tested for aliases. If the last character of the alias value is a blank, the word following the alias will also be checked for alias substitution.

Aliases can be used to redefine built-in commands but cannot be used to redefine the reserved words listed above. Aliases can be created, exported and listed on standard output with the `alias` command and can be

removed with the `unalias` command. Exported aliases remain in effect for scripts invoked by name, but must be reinitialized for separate invocations of the Korn shell.

Aliasing is performed when scripts are read, not while they are being executed. Thus for an alias to take effect the alias definition command has to be executed before the command which references the alias is read.

Aliases are frequently used as a short hand for full path names. An option to the built-in `alias` command allows the value of the alias to be automatically set to the full pathname of the corresponding command. These aliases are called tracked aliases. The value of a tracked alias is defined the first time the corresponding command is looked up and becomes undefined each time the `PATH` variable is reset. These aliases remain tracked so that the next reference will redefine the value. Several tracked aliases are compiled into the Korn shell. The `–h` option of the built-in `set` command makes each referenced command name into a tracked alias.

The following exported aliases are compiled into the Korn shell but can be unset or redefined with `unalias`.

```
autoload='typeset –fu'
false='let 0'
functions='typeset –f'
hash='alias –t'
history='fc –l'
integer='typeset –i'
nohup='nohup '
r='fc –e –'
true=':'
type='whence –v'
```

Examples:

The following aliases could be defined to construct `l`, `ll` and `lf`:

```
alias l='/bin/ls –m'
alias ll='/bin/ls –l'
alias lf='/bin/ls –CF'
```

Tilde substitution

After alias substitution is performed, each word is checked to see if it begins with an unquoted `~` (tilde). If it does, then the word up to the next / (slash) is checked to see if it matches a login name in the `/etc/passwd` file. If a match is found, the tilde and the matched login name are replaced by the login directory of the matched user. This is called a tilde substitution. If no match is found, the original text is left unchanged. A tilde by itself, or in front of a slash, is replaced by `$HOME`. A tilde followed by a plus sign or a minus sign is replaced by `$PWD` and `$OLDPWD`, respectively.

In addition, tilde substitution is attempted when the value of a "variable assignment" begins with a `~`.

Example:

The command

```
vi ~/.profile
```

allows you to edit the `.profile` file in your login directory from any location in your file tree.

Command substitution

There are two ways of substituting the standard output of a command for a part of a word or a full word. In the first (new) form, the command is enclosed in parentheses and preceded by a dollar sign `$(...)`. In the second (old) form, the command is enclosed within backquotes\p `...`. If the second form is used, the string between the backquotes is processed for special quoting characters before the command is executed (see **Quoting metacharacters** below). Trailing newlines are removed for both forms.

The command substitution `$(cat file)` can be replaced by the equivalent but faster `$(<file)`. Command substitutions on built-in commands which do not perform I/O redirection are carried out without a separate process being spawned.

Example:

The following command can be used to edit all the files in a directory which have names ending in `.c` and contain the string `include`:

```
vi $( grep -l include *.c )
```

`grep -l include *.c` scans all `.c` files for the string `include`. The `–l` option tells `grep` to list the names of all the files in which it finds the string.

An arithmetic expression enclosed in double parentheses and preceded by a dollar sign, i.e. `$((...))`, is replaced by the value of the arithmetic expression within the double parentheses.

Example:

This command will print the second-last parameter in a shell script:

```
eval print \$$(( $#-1 ))
```

Korn shell variables and parameter substitution

A parameter is an identifier, one or more digits, or any of the following characters:

`* @ # ? - $ !`

A variable (a parameter denoted by an identifier) has a value and zero or more attributes. Variables can be assigned values and attributes with the built-in `typeset` command. The attributes supported by the Korn shell are described under `typeset`. Exported parameters pass their values and attributes to the environment.

The Korn shell supports single-dimensional arrays. An element of an array variable is referenced by a subscript. A subscript is denoted by a left square bracket `[`, followed by an arithmetic expression (see **Arithmetic evaluation** below), followed by a right square brakket `]`.

To assign values to an array, you can use the built-in `set` command as follows:

`set –A name value ...`

The value of all subscripts must be in the permissible range. Arrays need not be declared; any reference to a variable with a valid subscript is legal, and an array will be created if necessary. Referencing an array without a subscript is equivalent to referencing element zero.

name=value [*name=value*] ...

The value of a variable may be assigned in this form. If the integer attribute is set for a variable, its value can be used for arithmetic evaluations. Positional parameters (denoted by a number) may be assigned values with the built-in `set` command. Parameter `$0` is set from argument zero when the shell is invoked.

${*parameter*}

The dollar sign `$` is used to introduce substitutable parameters. The Korn shell reads all the characters from `${` to the matching `}` as part of the same word even if it contains braces or metacharacters. The value, if any, of the parameter is substituted. The braces are required when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name, or when a variable is subscripted. If *parameter* is one or more digits, then it is a positional parameter. A positional parameter with more than one digit must be enclosed in braces.

If *parameter* is defined by the character `*` or `@`, then all the positional parameters, starting with `$1`, are substituted. The first character of the `IFS` variable is used as the field separator (see below). If an array identifier with the subscript `*` or `@` is used, the value for each of the elements is substituted (separated by the same field separator character).

${#*parameter*}

If *parameter* is `*` or `@`, the number of positional parameters is substituted. Otherwise, the length of the value of *parameter* is substituted.

${#*identifier*[*]}

The number of elements in the array *identifier* is substituted.

${*parameter*:-*word*}

If *parameter* is set and is non-null, its value is substituted; otherwise, *word* is substituted.

${*parameter*:=*word*}
If *parameter* is not set or is null, then it is set to *word*. The value of the parameter is then substituted.

Positional parameters may not be assigned values in this way.

${*parameter*:?*word*}
If *parameter* is set and is non-null, its value is substituted; otherwise, *word* is printed and the shell exits. If *word* is omitted, a standard message is printed.

${*parameter*:+*word*}
If *parameter* is set and is non-null, *word* is substituted; otherwise, the null string is substituted.

${*parameter*#*pattern*}
${*parameter*##*pattern*}
If the Korn shell *pattern* matches the beginning of the value of *parameter*, then the value of this substitution is the value of *parameter* with the matched portion deleted; otherwise, the value of *parameter* is substituted. In the first form, the smallest matching pattern is deleted; in the second form, the largest matching pattern is deleted.

${*parameter*%*pattern*}
${*parameter*%%*pattern*}
If the Korn shell *pattern* matches the end of the value of *parameter*, then the value of this substitution is the value of *parameter* with the matched portion deleted; otherwise, the value of *parameter* is substituted. In the first form, the smallest matching pattern is deleted; in the second form, the largest matching pattern is deleted.

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 *dir* is not set or is null:

```
echo ${dir:-$(pwd)}
```

If the colon is omitted from the above expressions, the Korn shell only checks whether or not *parameter* is set; it does not check whether it is equal to the null string.

The following parameters are automatically set by the Korn shell:

`#` The number of positional parameters (in decimal).

`–` All options supplied to the shell on invocation or by the `set` command.

`?` The exit status returned by the last executed command.

`$` The process ID of the current Korn shell.

`_` Initially, the value of _ (underscore) is an absolute pathname of the shell or script being executed as passed in the environment. Thereafter, it is always assigned the last argument of the previous command. This parameter is not set for commands which run asynchronously. The underscore parameter is also used to hold the name of the matching `MAIL` file when checking for mail.

Example:

```
cat /usr/tmp/mydir/xyz* > all
tail $_
```

The `tail` command accesses the last file of the previous `cat` command using `$_`.

`!` The process number of the last command invoked as a background process.

`ERRNO` The value of *errno* as set by the most recently failed system call. This value is system-dependent and is intended for debugging purposes.

`LINENO`
The line number of the current line within the script or function being executed.

`OLDPWD`
The previous working directory set by the `cd` command.

`OPTARG`
The value of the last option argument processed by the built-in `getopts` command.

`OPTIND`

The index of the last option argument processed by the built-in `getopts` command.

`PPID` The process number of the parent of the current Korn shell.

`PWD` The present working directory set by the `cd` command.

`RANDOM`
Each time this variable is referenced, a random integer, uniformly distributed between 0 and 32767, is generated. You can initialize the sequence of random numbers by assigning a numeric value to `RANDOM`.

`REPLY` This variable is set by the `select` statement and by the built-in `read` command when no arguments are supplied.

`SECONDS`
Each time this variable is referenced, the number of seconds since the invocation of the Korn shell is returned. If this variable is assigned a value, the value returned upon reference will be the value that was assigned plus the number of seconds since the assignment.

The following variables are used by the Korn shell:

`CDPATH`
The search path for the `cd` command.

`COLUMNS`
If this variable is set, its value is used to define the width of the edit window for the Korn shell edit modes and for printing `select` lists.

`EDITOR`
If the value of this variable ends in `emacs`, `gmacs`, or `vi` and the `VISUAL` variable is not set, the corresponding option will be turned on (see **Built-in commands** below, `set`).

`ENV` If this variable is set, parameter substitution is performed on its value to generate the pathname of the script that will be executed when the Korn shell is invoked (see **Invocation** below). This file is typically used for alias and function definitions.

`FCEDIT`
The name of the default editor for the built-in `fc` command.

`FPATH` The search path for function definitions. This path is searched when a function with the `-u` attribute is referenced and when a command is not found. If an executable file is found, it is read and executed in the current environment.

`IFS` The Korn shell's internal field separator, used to separate command words which result from command or parameter substitution and to separate words with the built-in `read` command. The value of `IFS` is normally set to space, tab, and newline. The first character of the `IFS` variable is used to separate arguments in `"$*"` substitutions (see **Quoting metacharacters** below).

`HISTFILE`
If this variable is set when the Korn shell is invoked, its value is the pathname of the file that will be used to store the command history (see **Command re-entry** below).

`HISTSIZE`
If this variable is set when the Korn shell is invoked, the shell will remember the commands you enter (command history). The number of previously entered commands that are accessible by this shell will be greater than or equal to the given number. The default is 128.

`HOME` The default argument (home directory) for the `cd` command.

`LINES` If this variable is set, its value is to determine the column length for printing `select` lists. `select` lists will print vertically until about two-thirds of `LINES` lines are filled.

`MAIL` If this variable is set to the name of a `mail` file and the `MAILPATH` variable is not set, the Korn shell informs you of the arrival of mail in the specified file.

`MAILCHECK`
The value of this variable specifies how often (in seconds) the Korn shell should check for changes in the modification time of any of the files specified by the `MAILPATH` or `MAIL` variables. The default value for `MAILCHECK` is 600 seconds. When the specified time has elapsed, the Korn shell checks the mail

files before issuing the next prompt.

MAILPATH
A colon-separated list of file names. If this variable is set, the shell informs you of any modifications to the specified files that have occurred within the last MAILCHECK seconds. Each file name may be followed in the list by a question mark and a message that is to be printed. The message will undergo parameter substitution, with the variable $_ defined as the name of the file that has changed. The default message is: "you have mail in $_".

PATH The search path for commands (see **Execution** below). You may not change the value of this variable if you are using a restricted Korn shell.

PS1 The value of this variable is expanded for parameter substitution to define the Korn shell's primary prompt string. The default is "$ ". An exclamation point in the primary prompt string will be replaced by the command number (see **Command re-entry** below).

PS2 Secondary prompt string displayed by the Korn shell when further input is expected after a newline character. The default is "> ".

PS3 Prompt string used within a select loop to prompt for the desired number. The default is "#? ".

PS4 The value of this variable is expanded for parameter substitution and precedes each line of an execution trace. The default execution trace prompt is "+ ".

SHELL The pathname of the Korn shell is kept in the environment. On Korn shell invocation, the shell will be made a restricted shell if the basename of this variable matches the pattern *r*sh* [see basename(1)].

TMOUT If TMOUT is set to a value greater than zero, the Korn shell will automatically terminate if a command is not entered within the prescribed number of seconds after issuing the PS1 prompt.

(Caution: the Korn shell may have been compiled with a maximum bound for this value which cannot be exceeded).

VISUAL
If the value of this variable ends in emacs, gmacs, or vi, the corresponding option will be turned on (see **Built-in commands** below, set).

The Korn shell assigns default values to the following variables:

PATH, PS1, PS2, MAILCHECK, TMOUT and IFS.

The HOME, MAIL and SHELL variables are set by login.

Blank interpretation

After parameter and command substitution, the results of substitutions are scanned for field separator characters and split into distinct arguments where such characters are found. Explicit null arguments (e.g. "" or '') are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.

File name generation

Following substitution, each command word is scanned for the asterisks (*), queries (?), and left square brackets ([), unless the –f option has been set (see set). If any of these characters appears in a word, the word is regarded as a pattern. The word is then replaced by lexicographically sorted file names that match the pattern. If no matching file name is found, the word is left unchanged. Note that when a pattern is used for file name generation, the dot character (.) at the start of a file name or immediately following a slash (/), as well as the slash character itself, must be matched explicitly. In other instances of pattern matching, the slash and the dot are not treated specially.

* Matches any string, including the null string.

? Matches any single character. (Slashes and dots are treated specially; see above.)

[...] Matches any one of the characters enclosed within the square brackets.

A pair of characters separated by a minus sign (–) matches any character lexically between the pair,

inclusive. A minus sign can be included in the set of characters to be matched as long as it is the first or last character.

If the first character following the left bracket is a an exclamation point, the enclosed set of characters is negated: any character not enclosed is matched.

A *pattern list* is a list of one or more patterns separated from each other by a | (vertical bar). Composite patterns can be formed with one or more of the following constructs:

`?` (*pattern-list*)
: Optionally matches any one occurrence of the given patterns.

`*` (*pattern-list*)
: Matches zero or more occurrences of the given patterns.

`+` (*pattern-list*)
: Matches one or more occurrences of the given patterns.

`@` (*pattern-list*)
: Matches exactly one of the given patterns.

`!` (*pattern-list*)
: Matches anything, except one of the given patterns.

Quoting metacharacters

Each of the metacharacters listed above (see **Definitions** above) has a special meaning to the Korn shell and acts as a word separator unless it is quoted. A single character may be quoted (i.e. made to stand for itself) by preceding it with a `\` (backslash). The `\newline` pair is ignored or removed by the Korn shell. All characters enclosed between a pair of single quotes (`'...'`) are quoted. A single quote cannot appear within single quotes. Inside double quotes (`"..."`), parameter and command substitution occurs, and a backslash quotes another backslash, a backquote (`` ` ``), a double quote (`"`), and a dollar sign (`$`).

The meaning of `$*` and `$@` is identical when not quoted or when used as a variable assignment value or as a file name. However, if they are enclosed in double quotes and used as a command argument, their meanings differ as follows: `"$*"` is equivalent to `"$1 $2 ..."`, where the is the first character of the `IFS` variable, whereas `"$@"` is equivalent to `"$1" "$2"...`, which means that individual command line arguments are preserved.

Inside backquotes, a backslash quotes another backslash, a backquote and a dollar sign. If the backquotes occur within double quotes, a backslash also quotes a double quote.

The special meaning of reserved words or aliases can be removed by quoting any character of the reserved word. The recognition of function names or built-in command names cannot be suppressed in this way.

Arithmetic evaluation

The built-in `let` command provides a mechanism for performing integer arithmetic. Evaluations are performed using `long` arithmetic. Constants are of the form [`base#`]*n*, where `base` is a decimal number between two and thirty-six representing the arithmetic base, and *n* is a number in that base. If `base#` is omitted, base 10 is used.

An arithmetic expression uses much the same syntax, precedence, and associativity as the C language. All the integral operators, other than `++`, `--`, `?:`, and the comma are supported. Variables can be referenced by name within an arithmetic expression without using the parameter substitution syntax (the $ character). When a variable is referenced, its value is evaluated as an arithmetic expression.

An internal integer representation of a variable can be specified as an attribute with the `-i` option of the built-in `typeset` command. Arithmetic evaluation is performed on the value of each assignment to a variable with the `-i` attribute. If you do not specify an arithmetic base, the first assignment to the variable determines the arithmetic base. This base is used when parameter substitution is performed.

Since many of the arithmetic operators must be quoted for the Korn shell, an alternative form of the `let` command is provided. For any command which begins with a double left parenthesis `((`, all the characters until a matching double right parenthesis `))` are treated as a quoted expression. Thus `((a=a+b))` is

equivalent to `let "a=a+b"`.

Conditional expressions

A conditional expression can be used to test attributes of files and to compare algebraic expressions and strings. In the Korn shell, conditional expressions are specified as part of a compound command of the form `[[...]]`. Blank interpretation and file name generation are not performed on the words of the conditional expression between `[[` and `]]`. Each conditional expression can be constructed from one or more of the following unary or binary expressions:

File attributes

In each of the expressions below, if *file* is of the form `/dev/fd/`*n*, where *fd* is the file descriptor and *n* is an integer, then the test is applied to the open file whose descriptor number is *n*.

`-a` *file*
: (access) True if *file* exists.

`-b` *file*
: (block device) True if *file* exists and is a block special file.

`-c` *file*
: (character device) True if *file* exists and is a character special file.

`-d` *file*
: (directory) True if *file* exists and is a directory.

`-f` *file*
: (file) True if *file* exists and is an ordinary file.

`-g` *file*
: (group ID) True if *file* exists and has its set-group-ID bit set.

`-k` *file*
: (sticky) True if *file* exists and is has its sticky bit set.

`-o` *option*
: (option) True if the named *option* is turned on (*option* can be set with `set`). You must use the full option name, e.g. *errexit*.

`-p` *file*
: (pipe) True if *file* exists and is a FIFO special file or a pipe.

`-r` *file*
: (read) True if *file* exists and the current process has read permission for it.

`-s` *file*
: (size) True if *file* exists and has a size greater than zero.

`-t` *fildes*
: (terminal) True if file descriptor number *fildes* is open and is associated with a terminal.

`-u` *file*
: (user ID) True if *file* exists and has its set-user-ID bit set.

`-w` *file*
: (write) True if *file* exists and the current process has write permission for it.

`-x` *file*
: (execute) True if *file* exists and the current process has execute permission for it. If *file* exists and is a directory, then the current process must have permission to search in the directory.

`-G` *file*
: (group) True if *file* exists and its group matches the effective group ID of the current process.

`-L` *file*
: (symbolic link) True if *file* exists and is a symbolic link.

`-O` *file*

(owner) True if *file* exists and is owned by the effective user ID of the current process.

`-S` *file*
(socket) True if *file* exists and is a socket.

file1 `-nt` *file2*
(newer than) True if *file1* exists and is newer than *file2*.

file1 `-ot` *file2*
(older than) True if *file1* exists and is older than *file2*.

file1 `-ef` *file2*
(equal file) True if *file1* and *file2* exist and are links to the same file.

String attributes and comparisons

`-n` *string*
(non-zero) True if *string* exists and is not a null string, i.e. if the length of the string is greater than 0.

`-z` *string*
(zero) True if the specified *string* is a null string, i.e. if the length of the string is 0.

string `=` *pattern*
True if *string* matches *pattern*.

string `!=` *pattern*
True if *string* does not match *pattern*.

string1 `<` *string2*
True if *string1* comes before *string2* in the ASCII collating sequence.

string1 `>` *string2*
True if *string1* comes after *string2* based in the ASCII collating sequence.

Algebraic comparisons between integers

expr1 `-eq` *expr2*
(equal) True if *expr1* is equal to *expr2*.

expr1 `-ne` *expr2*
(not equal) True if *expr1* is not equal to *expr2*.

expr1 `-lt` *expr2*
(less than) True if *expr1* is less than *expr2*.

expr1 `-gt` *expr2*
(greater than) True if *expr1* is greater than *expr2*.

expr1 `-le` *expr2*
(less than or equal) True if *expr1* is less than or equal to *expr2*.

expr1 `-ge` *expr2*
(greater than or equal) True if *expr1* is greater than or equal to *expr2*.

Negated and compound expressions

A compound expression can be constructed from the above expressions with any of the following mechanisms. The mechanisms are listed in decreasing order of precedence:

`(` *expression* `)`
True if *expression* is true. The enclosed *expression* can be a single expression or a group of concatenated expressions.

`!` *expression*
Negation: true if *expression* is false.

expression1 `&&` *expression2*
Logical AND: True if *expression1* and *expression2* are both true.

expression1 | | *expression2*
Logical OR: True if either *expression1* or *expression2* is true.

I/O redirection

Before a command is executed, its input and output may be redirected by means of a special notation interpreted by the Korn shell. The following arguments may appear anywhere in a simple command or may precede or follow a command. They are not passed on to the invoked command, but are interpreted by the shell. Command and parameter substitution occurs before the specified file or file descriptor is used. File name generation is performed only if the pattern matches a single file. Blank interpretation is not performed.

<*file* Redirects the standard input (file descriptor 0) of a command to *file*, so that input to the command is read from *file*.

>*file* Redirects the standard output (file descriptor 1) of a command to *file*, so that output from the command is written to *file*. If the file does not exist, it is created. If it does exist, is a regular file, and the `noclobber` option is set (see **Built-in commands** below, `set -o`), this causes an error. If `noclobber` is not set, the file is truncated to zero length and its existing contents are lost.

>|*file*
Same as >*file*, except that it overrides the `noclobber` option.

>>*file*
Redirects the standard output (file descriptor 1) of a command to *file*, so that output from the command is written to *file*. If the file exists, output is appended to it; otherwise, the file is created.

<>*file*
Opens *file* for reading and writing as standard input.

<<[–]*string*
Introduces a "here document" (explained below). No parameter substitution, command substitution or file name generation is performed on *string*. Input to the Korn shell is read up to (but excluding) a line that literally matches *string* or to an end-of-file. The resulting document, called a "here document", becomes the standard input for the command.

If any character of *string* is quoted, all characters in the here document are treated as quoted.

Caution:

The *string* marking the last line must not be quoted.

If no characters of *string* are quoted:

- parameter and command substitution is performed,
- escaped newlines (`\newline`) are ignored, and
- abackslash must be used to quote backslashes, dollar signs, backquotes and the first character of *string* if they are to be treated literally in the text.

If – is appended to <<, then all leading tabs are stripped from *string* and from each line in the here document.

<&*digit*
>&*digit*
In the first form, the standard input is duplicated from file descriptor *digit*, i.e. is read from the file associated with *digit*. The second form does the same for the standard output.

<&-
>&-
The first form closes the standard input for the command, so that EOF is the only input to the command. The second form closes the standard output for the command, so that the command does not write any output.

<&p
>&p
The input from the co-process in a two-way pipeline is redirected to standard input. Similarly, the output

to the co-process is redirected to standard output.

If one of the above redirections is preceded by a digit, the file descriptor number referred to is that specified by the digit (instead of the default 0 for `stdin` and 1 for `stdout`).

Example:

To open file descriptor 2 (`stderr`) for writing as a duplicate of file descriptor 1 (`stdout`):

```
... 2>&1
```

The order in which redirections are specified is significant. The Korn shell evaluates each redirection in terms of the association (file descriptor, file) at the time of evaluation. For example:

```
... 1>file 2>&1
```

first associates standard output (file descriptor 1) with *file* and then associates standard error (file descriptor 2) with the file associated with file descriptor 1 (i.e. *file*).

In other words, both the standard output and the error messages from the command are written to *file*.

If the order were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been), file descriptor 1 with *file*. This would mean that only the standard output would be written to file, not the error messages.

If a command is started in the background with `&` and job control is not active, the default standard input for the command is the empty file `/dev/null`. If job control is active, the environment for the execution of a command contains the file descriptors of the invoking Korn shell as modified by input/output specifications.

You can set commands or pipelines to run in the background from within a script. They can then communicate with your program. To spawn a co-process of this type, you put the operator `&` after the command. You can only use two-way pipes in scripts, not on the command line.

A group of commands can only be called once as a two-way pipe. If you terminate the original process (e.g. using `kill -9 PID`) and then later again try to write a command to a two-way pipe, a subshell will be called, but the process will be stopped and the error message "ksh: bad file unit number" will be issued.

EXECUTION ENVIRONMENT

Environment

The environment of a process consists of a list of *name=value* pairs that is passed to an executed program in the same way as a normal argument list. The names must be Korn shell-style identifiers; the values must be character strings (including the null string). The Korn shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a variable for each name found, giving it the corresponding value and marking it for export. Executed commands inherit the environment. If the user modifies the values of these variables or creates new ones using the `export` or `typeset -x` commands, they become part of the environment. The environment seen by any executed command is thus composed of any *name=value* pairs originally inherited by the shell, whose values may be modified by the current shell, plus any additions that have been marked in `export` or `typeset -x` commands.

The environment for any simple command or function may be augmented by prefixing it with one or more variable assignments. A variable assignment argument is a word of the form *identifier=value*.

command sees the following lines as equivalent:

```
TERM=450 command arguments
(export TERM; TERM=450; command arguments)
```

(Some built-in commands deviate from this rule. These commands are marked `+` or `++` in the section on **Built-in commands** below.)

If the `-k` option was set on Korn shell invocation or with the built-in `set` command, all variable assignment arguments are exported to the environment, even if they occur after the command name.

Example:

The following example first prints `a=b c` and then `c`:

```
echo a=b c
set –k
echo a=b c
```

This feature is intended for use with scripts written for early versions of the Bourne shell. It is best not to use it in new scripts, since the `–k` option may no longer be supported in future releases.

Functions

The reserved word `function` (see **Compound commands** above) is used to define Korn shell functions. Shell functions are read in and stored internally. Alias names are resolved when the function is read. Functions are executed like commands, with arguments passed as positional parameters (see **Execution** below).

Functions execute in the same process as the caller and share all open files and the present working directory with the caller.

Traps caught by the caller are reset to their default action inside the function. A trap condition that is not caught or ignored by the function causes the function to terminate; the condition is then passed on to the caller. A trap on `EXIT` set inside a function is executed after completion of the function in the environment of the caller.

Variables are usually shared between the calling program and the function. However, the built-in `typeset` command can be used within a function to define local variables whose scope includes the current function and all functions it calls.

The built-in `return` command is used to return from function calls. Errors that occur within functions return control to the caller.

Function identifiers or names can be listed with the `–f` or `+f` option of the built-in `typeset` command. The text of functions may also be listed with the `–f` option. Functions can be undefined with the `–f` option of the built-in `unset` command.

Functions are usually not accessible when the Korn shell executes a shell script.

The `–xf` option of the `typeset` command allows a function to be marked for export. These functions can then be used in scripts that are executed without a separate invocation of the Korn shell. Functions that need to be defined across separate invocations of the shell should be specified in the `ENV` file with the `–xf` option of `typeset`.

Example:

The following function named `lh` lists the top two levels of the directory hierarchy in which you are located or which you specify as an argument. Regular files are ignored as arguments.

```
function lh
{
  for i in ${*:-.}
  do
     if [[ -d $i ]]
     then
        print $i:
        cd $i
        ls -CF $( ls )
        cd - >/dev/null
     fi
  done
}
```

The `for` loop processes each of the specified arguments in succession. If you have not specified an argument, `$*` is set to the current directory (`.`).

If a directory is present, its name and contents are listed as follows: `cd $i` switches to the directory to be

listed; `$(ls)` is replaced by the content of the directory `$i`, and `ls -CF` *directory_list* lists the files, followed by the contents of the directories, with flags to indicate executables and directories. The subsequent `cd` command silently returns to the initial directory (before the first `cd $i`).

Jobs

If the `monitor` option of the built-in `set` command is turned on, an interactive Korn shell associates a job with each pipeline. It maintains a table of current jobs, which you can written to standard output with the built-in `jobs` command, and assigns them small integer numbers. When a job is started asynchronously with `&`, the shell prints a line of the form:

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process ID was 1234.

If one job is running and you want to start other jobs, you can simply press <CTRL-Z>. This sends a `STOP` signal to the current job. The shell will then normally indicate that the job has been "Stopped", and will issue another prompt. You can then manipulate the state of this job by putting it in the background with the `bg` command, or you can leave it in the stopped state, run some other commands and then eventually bring the job back into the foreground with the foreground command `fg`.

<CTRL-Z> takes effect immediately and is like an interrupt with <CTRL-D> in that pending output and unread input are discarded when it is typed.

A background job will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but you can stop this with the command `stty tostop`. If you set this `stty` option, background jobs will stop when they try to produce output.

There are several ways to reference jobs in the shell. A job can be referenced by the process ID of any of its processes or by one of the following expressions:

`%`*number*
: the job with the given job *number*

`%`*string*
: any job whose command line begins with *string*

`%?`*string*
: any job whose command line includes *string*

`%%`
: the current job

`%+`
: equivalent to `%%`

`%-`
: the previous job

The Korn shell registers changes in the status of jobs immediately. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work.

When the `monitor` option is turned on, each background job that completes triggers any trap set for `CHLD`.

If you try to exit the Korn shell while jobs are running in the background or stopped, you will be warned with the message: `You have stopped (running) jobs.`

You may then use the `jobs` command to obtain an overview of the current situation. If you do this or immediately try to exit again, the shell will not warn you a second time, and the stopped jobs will be terminated.

Signals

The `INT` and `QUIT` signals for an invoked background command (`&`) are ignored if the job `monitor` option is not active. Otherwise, signals have the values inherited by the Korn shell from its parent (see also the built-in `trap` command below).

Execution

Each time a command is executed, the substitutions described above are carried out in the following order:

- Quoting
- Parameter substitution
- Tilde substitution
- Aliasing
- File name generation
- I/O redirection
- Command substitution

If the command name matches the name of one of the built-in commands, it is executed within the current Korn shell process. Next, the command name is checked to see if it matches one of the user-defined functions. If it does, the positional parameters are saved and then reset to the arguments of the function call. When the function completes or executes a built-in `return` command, the positional parameter list is restored and any trap set on `EXIT` within the function is executed. The exit status is that of the last command executed in the function. A function is also executed in the current Korn shell process. If a command name is not a built-in command or a user-defined function, a process is spawned and an attempt is made to execute the command via the system call `exec`.

The shell variable `PATH` defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is `/usr/bin:` (specifying `/usr/bin` and the current directory in that order). The current directory can be specified by two or more adjacent colons, or by a colon at the beginning or end of the path list. If the command name contains a slash, the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not a directory or an `a.out` file, it is assumed to contain a shell script. A subshell is spawned to read it. Any non-exported aliases, functions and variables are removed in this case. A parenthesized command is executed in a sub-shell without removing non-exported quantities.

COMMAND PROCESSING

Command re-entry

The text of the last `HISTSIZE` commands entered from a terminal is saved in a `history` file. The default value for `HISTSIZE` is 128. If the `HISTFILE` variable is not set or if the file denoted by its value is not writable, the file `$HOME/.sh_history` is used to save the command history. A Korn shell can access the commands of all interactive shells which use the same history file. The built-in `fc` command can be used to list or edit a portion of this file. The portion of the file to be edited or listed can be selected by number or by giving the first character or characters of the command. A single command or range of commands may be specified. If you do not specify an editor program as an argument to `fc`, the value of the variable `FCEDIT` is used. If `FCEDIT` is not defined, `/usr/bin/ed` is used. The edited commands are printed and re-executed when you leave the editor.

Specifying `-s` skips the edit phase. In this case, a variable of the form *old=new* can be used to modify the command before it is executed. If, for example, you enter `fc -s nonsense=sense c`, the last command that began with the character `c` is executed. Before this takes place however, the first occurrence of `nonsense` in the command is replaced by `sense`.

In-line editing options

Normally you simply type in a command line and terminate it with a newline character. However, if the `vi` option is active (see `set` below), you can also edit command lines.

This editing option is automatically selected when the `VISUAL` or `EDITOR` variable is assigned a value that ends with `emacs`, `gmacs`, or `vi`.

The editing features require the terminal to have the following capabilities:

- it must accept "return" as a carriage return without a line feed, and
- aspace ( ) must overwrite the current character on the screen.

ADM terminal users should set the "space – advance" switch to "space". Hewlett-Packard series 2621 terminal users should set the straps to "`bcGHxZ etX`".

The editing mode uses the concept of a window on the current line. The window width is the value of `COLUMNS` if it is defined, or 80 characters if it is not. If the line is longer than the window width minus two, you are notified by a mark displayed at the end of the window. As you move the cursor across a window boundary, the window moves to center about the cursor position. The following marks are used:

`>` if the line extends beyond the right side of the window

`<` if the line extends beyond the left side of the window

`*` if the line extends beyond both sides of the window

The search commands in each edit mode provide access to the history file. Only strings are matched, not patterns, except that a caret (`^`) as the first character of a string restricts the match to begin at the first character in the line.

emacs editing mode

This mode is entered by enabling either the `emacs` or `gmacs` option. The only difference between these two modes is the way they handle `^T`. To edit, the user moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. All the editing commands are control characters or escape sequences. The notation for control characters is caret (`^`) followed by the character. For example, `^F` is the notation for <CTRL-f>. This is entered by depressing "f" while holding down the "CTRL" (control) key. The "SHIFT" key is *not* depressed. (The notation `^?` indicates the DEL (delete) key.)

The notation for escape sequences is `M-` followed by a character. For example, `M-f` (pronounced Meta f) is entered by depressing ESC (ASCII \033) followed by "f". (`M-F` would be the notation for ESC followed by "SHIFT" (capital) "F".)

All edit commands operate from any place on the line (not just at the beginning). Neither the "RETURN" nor the "LINEFEED" key is entered after edit commands except when noted.

`^F` Move cursor forward (right) one character.

`M-f` Move cursor forward one word. (The `emacs` editor's idea of a word is a string of characters consisting of only letters, digits and underscores.)

`^B` Move cursor backward (left) one character.

`M-b` Move cursor backward one word.

`^A` Move cursor to start of line.

`^E` Move cursor to end of line.

`^]`*char*
Move cursor forward to character *char* on current line.

`M-^]`*char*
Move cursor back to character *char* on current line.

`^X^X` Interchange the cursor and mark.

erase (User defined erase character as defined by the `stty(1)` command, usually `^H` or `#`.) Delete previous character.

`^D` Delete current character.

`M-d` Delete current word.

`M-^H` (Meta-backspace) Delete previous word.

`M-h` Delete previous word.

`M-^?` (Meta-DEL) Delete previous word (if your interrupt character is `^?` (DEL, the default) then this command will not work).

`^T` Transpose current character with next character in `emacs` mode. Transpose two previous characters in `gmacs` mode.

^C Capitalize current character.

M-c Capitalize current word.

M-l Change the current word to lower case.

^K Delete from the cursor to the end of the line. If preceded by a numerical parameter whose value is less than the current cursor position, then delete from given position up to the cursor. If preceded by a numerical parameter whose value is greater than the current cursor position, then delete from cursor up to given cursor position.

^W Kill from the cursor to the mark.

M-p Push the region from the cursor to the mark on the stack.

kill (User defined kill character as defined by the stty command, usually ^G or @.) Kill the entire current line. If two kill characters are entered in succession, all kill characters from then on cause a line feed.

^Y Restore last item removed from line. (Yank item back to the line.)

^L Line feed and print current line.

^@ (Null character) Set mark.

M-*space*
(Meta space) Set mark.

^J (Newline) Execute the current line.

^M (Return) Execute the current line.

eof End-of-file character, normally ^D, is processed as an end-of-file only if the current line is null.

^P Fetch previous command. Each time ^P is entered the previous command back in time is accessed. Moves back one line when not on the first line of a multi-line command.

M-< Fetch the least recent (oldest) history line.

M-> Fetch the most recent (youngest) history line.

^N Fetch next command line. Each time ^N is entered the next command line forward in time is accessed.

^R*string*
Reverse search history for a previous command line containing *string*. If a parameter of zero is given, the search is forward. *string* is terminated by a "RETURN" or "NEWLINE". If string is preceded by a ^, the matched line must begin with *string*. If *string* is omitted, then the next command line containing the most recent *string* is accessed. In this case a parameter of zero reverses the direction of the search.

^O Operate – Execute the current line and fetch the next line relative to current line from the history file.

M-*digits*
(Escape) Define numeric parameter, the digits are taken as a parameter to the next command. The commands that accept a parameter are ^F, ^B, *erase*, ^C, ^D, ^K, ^R, ^P, ^N, ^], M-., M-^], M-_, M-b, M-c, M-d, M-f, M-h, M-l and M-^H.

M-*letter*
Soft-key – Your alias list is searched for an alias by the name _*letter* and if an alias of this name is defined, its value will be inserted on the input queue. The *letter* must not be one of the above meta-functions.

M-] *letter*
Soft-key – Your alias list is searched for an alias by the name __*letter* and if an alias of this name is defined, its value will be inserted on the input queue. The can be used to program functions keys on many terminals.

M-. The last word of the previous command is inserted on the line. If preceded by a numeric parameter, the value of this parameter determines which word to insert rather than the last word.

M-_ Same as M-..

`M-*` Attempt file name generation on the current word. An asterisk is appended if the word doesn't match any file or contain any special pattern characters.

`M-ESC` File name completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. If the match is unique, a `/` is appended if the file is a directory and a space is appended if the file is not a directory.

`M-=` List files matching current word pattern if an asterisk were appended.

`^U` Multiply parameter of next command by 4.

`\` Escape next character. Editing characters, the user's erase, kill and interrupt (normally `^?`) characters may be entered in a command line or in a search string if preceded by a `\`. The `\` removes the next character's editing features (if any).

`^V` Display version of the shell.

`M-#` Insert a `\#` at the beginning of the line and execute it. This causes a comment to be inserted in the history file.

vi editing mode

There are two typing modes:

- When you enter a command you are in *input* mode.
- When you press the <ESCAPE> key, you enter *control* mode.

In this mode you can edit by moving the cursor to the point needing correction and then inserting or deleting characters. As with `vi`, most commands accept an optional repeat count prior to the command.

The initial setting on most systems in `vi` mode is input mode. Entering <ESCAPE> terminates the input mode and allows you to modify the command line. This scheme offers the combined advantages of input mode as an initial setting and the type-ahead echoing of raw mode.

Input edit commands

By default the editor is in input mode.

<BACKSPACE>
<BACKSPACE> represents the `erase` character as defined by the `stty` command.) Deletes the character to the left of the cursor.

<CTRL-W>
Deletes the previous blank-delimited word.

<CTRL-D>
Terminates the Korn shell, unless the `ignoreeof` option is set (see `set` below).

<CTRL-V>
Escapes the next character. Editing characters and the `erase` and `kill` characters may be entered in a command line or in a search string if preceded by a <CTRL-V>. The <CTRL-V> removes the next character's editing features (if it has any).

`\` Escapes the next `erase` or `kill` character.

Motion edit commands

These are commands for moving the cursor. A *count* is equivalent to repeating the command the specified number of times.

[*count*]`l`
Cursor one character forward (right).

[*count*]`w`
Cursor one alphanumeric word forward (right).

[*count*]`W`
Cursor to the beginning of the next word that follows a blank.

[*count*]`e`

Cursor to end of word.

[*count*]E
: Cursor to end of the current blank-delimited word.

[*count*]h
: Cursor one character back (left).

[*count*]b
: Cursor one word back (left).

[*count*]B
: Cursor to preceding blank-delimited word.

[*count*]|
: Cursor to column *count*.

[*count*]f*c*
: Finds the next occurrence of character *c* in the current line.

[*count*]F*c*
: Finds the previous occurrence of character *c* in the current line.

[*count*]t*c*
: Equivalent to f followed by h.

[*count*]T*c*
: Equivalent to F followed by l.

[*count*];
: Repeats *count* times the last single-character find command given with f, F, t or T.

[*count*],
: Reverses the last single-character find command *count* times.

0 Cursor to start of line.

^ Cursor to first visible non-blank character in line.

$ Cursor to end of line.

Search edit commands

These are commands for accessing your command history file:

[*count*]k
: Fetches the previous command. Each time k is entered the previous command back in time is accessed.

[*count*]–
: Equivalent to k.

[*count*]j
: Fetches the next command. Each time j is entered the next command forward in time is accessed.

[*count*]+
: Equivalent to j.

[*count*]G
: Fetches command number *count*. The default is the least recent history command.

/*string*
: Searches backward through the history file for an earlier command including *string*. *string* is terminated by a newline character. If *string* is preceded by a caret (^), the matched line must begin with *string*. If *string* is null, the previous *string* will be used.

?*string*
: Same as /, but searching forward.

n Searches for the next line matching the last search string.

N Search for next line matching the last search string, but in reverse direction. Searches the history file

for the *string* entered with the previous / command.

Text modification edit commands

These are commands for modifying the command line.

a Enters input mode and inserts text after the current character.

A Appends text to the end of the line. Equivalent to $a.

[*count*]c*motion*
c[*count*]*motion*
Deletes the current character through the character that *motion* would move the cursor to and enters input mode. If *motion* is c, the entire line is deleted and input mode is entered.

C Deletes the current character through the end of the line and enters input mode.

S Equivalent to cc.

D Deletes the current character through the end of the line. Equivalent to d$.

[*count*]d*motion*
d[*count*]*motion*
Deletes the current character through the character that *motion* would move to. If *motion* is d, the entire line is deleted.

i Enters input mode and inserts text before the current character.

I Inserts text before the beginning of the line. Equivalent to 0i.

[*count*]P
Places (*count* copies of) the previous text modification before the cursor.

[*count*]p
Places (*count* copies of) the previous text modification after the cursor.

R Enters input mode and overwrites existing text from the current character onward.

[*count*]r*character*
Replaces the next *count* character(s) starting at the current cursor position with *character* and advances the cursor to the last replaced character.

[*count*]x
Deletes the current character (and the next *count* minus 1 characters).

[*count*]X
Deletes the preceding (*count*) character(s).

[*count*].
Repeats the previous text modification command (*count* times).

[*count*]~
Inverts the case of the (*count*) character(s) starting at the current cursor position and advances the cursor *count* characters.

[*count*]_
Appends the last word (or word number *count*) of the previous command and enters input mode.

* Attempts file name generation with the current word by appending an asterisk to it. If no match is found, the terminal bell rings. Otherwise, the word is replaced by the matching file name and input mode is entered.

\ Filename completion: replaces the current word with the longest common prefix of all file names matching the current word with an asterisk appended. If the match is unique, a slash is appended if the file is a directory and a space is appended if the file is not a directory.

Other edit commands

[*count*]y*motion*
y[*count*]*motion*

Yanks (copies) the current character through the character that *motion* would move the cursor to and puts them in the delete buffer. The text and the cursor position are left unchanged.

`Y` Yanks text from the current character position to the end of the line. Equivalent to `y$`.

`u` Undoes the last text modification command.

`U` Undoes all the text modification commands performed on the command line.

[*count*]`v`
Reads *count* lines from the history file into the input buffer using the command:
`fc –e ${VISUAL:-$(EDITOR:-vi}} count`.
If *count* is omitted, the current line is used.

<CTRL-L>
(line feed) Outputs a line feed and prints the current line. This command only works in command mode.

<CTRL-J>
(newline) Executes the current line, regardless of the current mode.

<CTRL-M>
(carriage return) Executes the current line, regardless of the current mode.

`#` Sends the line after inserting a `#` (hash character) in front of the line. Useful for causing the current line to be inserted in the history without being executed.

`=` Lists the file names that would match the current word if an asterisk were appended to it.

`@`*letter*
Searches the alias list for an alias named *_letter*, and if it finds one, uses its value as input.

Built-in commands

In Korn shell documentation, built-in commands are sometimes referred to as special commands. The term "built-in command" is preferred here for consistency with `csh` and `sh`.

The following simple commands are executed in the Korn shell itself. Input/output redirection is permitted. Unless otherwise indicated, the output is written to standard output, and the exit status is zero unless there is a syntax error.

Commands that are preceded by one or two plus characters (+ or ++) in the description are treated specially in the following ways:

- Variable assignment lists preceding the command name remain in effect when the command completes.
- I/O redirections are processed after variable assignments.
- Errors cause a script that contains them to abort.
- ++ only:

  Words that are in the form of a variable assignment (*name=value*) are expanded with the same rules as a variable assignment.

  This means that tilde substitution is performed after the equal sign, but word splitting (by blank interpretation) and file name generation are not.

The commands with + are: `:` (colon), `.` (dot), `break`, `continue`, `eval`, `exec`, `exit`, `newgrp`, `return`, `shift`, `times`, `trap`, `wait`.

The commands with ++ are: `alias`, `export`, `readonly`, `typeset`.

`+:` [*arg* ...]
This command simply expands parameters in *arg*.

`+.` *file* [*arg* ...]
Reads the entire *file* and executes the commands it contains in the current Korn shell environment. The search path specified by `PATH` is used to find the directory containing *file*.

If any arguments *arg* are given, they become the positional parameters; otherwise, the positional parameters are unchanged. The exit status is the exit status of the last command executed.

`[` Test conditions, see `test(1)` .

`++alias` [`–t`] [*name*]
`alias` with no arguments writes the list of aliases in the form *name=value* to standard output.

`–t` Used to set and list tracked aliases. The *value* of a tracked alias is the full pathname corresponding to the given *name*. The value becomes undefined when the value of `PATH` is reset, but the aliases remain tracked. Without the `–t` option, the *name=value* pair is printed for each name in the argument list for which no value is given.

`++alias` [`–x`] [*name*=[*value*]]...
Aspace at the end of *value* causes the next word in the command line to be checked for alias substitution.

`–x` Used to set or list exported aliases. An exported alias is defined for scripts invoked by name.

The exit status is non-zero if no *value* is defined for *name*.

`bg` [*job* ...]
Moves each specified job into the background. If *job* is not specified, the last stopped job is moved into the background. The format of *job* is as described under **Jobs** above.

`+break` [*num*]
Exits from the enclosing `for`, `while`, `until` or `select` loop. If *num* is specified, execution is resumed in the *num*th enclosing loop. `break 0` causes an exit from the outermost loop. `break 1` is equivalent to `break`.

`cd` [*dir*]

`cd` *old new*

Format 1: Change directory using `CDPATH`

`cd` changes the current directory to *dir*. If you specify a minus sign (`–`) for *dir*, the directory is changed to the previous directory. The `PWD` variable is set to the current directory, and `OLDPWD` is set to the previous value of `PWD`. The shell variable `CDPATH` defines the search path for the directory containing *dir*.

As in the case of `PATH`, alternative directory names listed in `CDPATH` are separated by colons (:). The default path is the null string, which corresponds to the current directory. Note that the current directory is specified by a null path name, which can appear anywhere in `CDPATH`, even after the equal sign. If *dir* begins with a slash, the search path is not used. Otherwise, each directory in the search path is searched for *dir* in the given order.

Format 2: Change directory with text substitution

`cd` substitutes the string *new* for the string *old* in the current directory name (`PWD`) and tries to change to this new directory.

`+continue` [*num*]
Resumes execution at the next iteration of the enclosing `for`, `while`, `until` or `select` loop. If *num* is specified, then execution is resumed at the *num*th enclosing loop. `continue 0` resumes execution after the outermost loop. `continue 1` is equivalent to `continue`.

`command` [`–p`] *commandname* [*argument* ...]

`command` [`–v`| `–V`] *commandname*

Refer to the description of the `command(1)` command.

`echo` [*arg* ...]
Refer to the description of the `echo(1)` command.

`+eval` [*arg* ...]
The arguments are read as input to the Korn shell, and the resulting commands are executed. Thus there are two stages of argument evaluation.

`+exec` [*arg* ...]

`exec` [*redirection* ...]

Format 1: Overlay Korn shell

If *arg* is given, the command specified by the arguments is executed in place of the current Korn shell without a new process being spawned. I/O redirection, if specified, is carried out by the current Korn shell and then applies to the executed command.

Format 2: I/O redirection

If *redirection* is given, the effect of this command is to modify file descriptors as described in the section on **I/O redirection** above. In this case, any file descriptor numbers greater than 2 that are opened with this mechanism are closed when another program is invoked.

`+exit` [*num*]

This command terminates the Korn shell with the exit status specified by *num*. If *num* is omitted, the exit status is that of the last command executed. An end-of-file will also cause the shell to exit, provided the `ignoreeof` option has not been set (see `set` below).

`++export` [*name*=[*value*]]...□

`export` [`–p`]

The given names are marked for automatic export to any new environment. If `export` is invoked without arguments, all variables marked for exported are listed on standard output. `–p` generates a portable format with export statements, that can be used to backup and restore a particular status.

`false` See `let 0` or the `false(1)` command.

`fc` [`–e` *ename*] [`–lnr`] [`––`] [*first* [*last*]]

`fc` [`–s`] [`––`] [*old*=*new*] [*commando*]

Format 1: Edit commands from the history file

The range of commands from *first* to *last* in the last `HISTSIZE` commands typed at the terminal is selected for editing and subsequent execution. The *first* and *last* arguments may be specified as numbers or as strings.

A positive number stands for the number of the command; a negative number is interpreted as an offset from the current command number.

A string is used to locate the most recent command starting with the given string. If *last* is not specified, it is set to *first*; if *first* is not specified either, the default is the previous command for editing and `–16` for listing.

`–e` *editor*

editor is the name of the editor to be used. If the editor name is not supplied, the value of the variable `FCEDIT` is used.

If `FCEDIT` is not set, the default editor `/usr/bin/ed` is invoked.

`–l` Lists the specified commands on standard output. The commands are not edited or executed. `–l` cannot be combined with `–e`.

`–n` Suppresses command numbers when listing commands with the `–l` option.

`–r` Reverses the order in which the commands are presented.

`––` If the first argument begins with a minus sign (`–`), the end of the command line options must be marked with `––`. .RE

Format 2: Re-execute commands

The command is re-executed with *new* substituted for *old*.

`–s` The commands are executed without calling an editor first.

`fg` [*job* ...]

Moves each job specified is into the foreground. If no *job* is specified, The last job that was stopped or moved into the background is moved into the foreground. The format of *job* is as described under **Jobs** above.

`getopts` *optstring name* [*arg* ...]

`getopts` checks *arg* for legal options. If *arg* is omitted, the positional parameters are used. An option argument begins with a plus or minus sign. The options are ended by an argument not beginning with a plus or minus sign, or by the argument —. *optstring* contains the letters that `getopts` recognizes as options. If a letter is followed by a colon, that option is expected to have an argument. The options can be separated from the arguments by blanks.

Each time `getopts` is invoked, it assigns the next option letter it finds to the variable *name*, with a plus sign prepended to it if the option begins with a plus sign. The index of the next argument is stored in `OPTIND`. The option argument, if any, is stored in `OPTARG`.

A colon at the start of *optstring* causes `getopts` to store the letter of an invalid option in `OPTARG`, and to set *name* to `?` for an unknown option, or to `:` when a required option is missing. Otherwise, `getopts` prints an error message. The exit status is non-zero when there are no more options.

`jobs` [`–lnp`] [*job* ...]

This command is only available on systems that support job control. It lists information (sequence, status, and calling command) about each given job, or about all active jobs if *job* is omitted.

`–l` Lists process IDs in addition to the normal information.

`–n` Displays only those jobs that have stopped or exited since the last invocation of the `jobs` command.

`–p` Lists only the process group.

See the **Jobs** section above for a description of the format of *job*.

`kill` [`–s` *signal*] *job* ...

`kill –l` [*number*]

Format 1: Send signals to processes

Sends the `TERM` signal or the signal specified as `signal` to the job or process identified by *job*. Signals are defined by name (as listed in the file `/usr/include/signal.h`, without the `SIG` prefix). If the `TERM` or `HUP` signal is to be sent to a stopped job or process, a `CONT` signal will be sent to it first. The *job* argument can be the process ID of a process that is not a member of one of the active jobs.

The format of *job* is as described under **Jobs** above.

Format 2: List signal numbers and names

`kill -l` lists all signal numbers and their symbolic names on the standard output, one per line. If *number* is specified, only the signal with the corresponding signal number is output.

`let` *argument* ...

Each argument is an arithmetic expression. The results obtained on evaluation are output. The exit status is 0 if the value of the last expression is non-zero, and 1 otherwise.

`login` [*login_name* [*environment* ...]]

Refer to the description of the `login(1)` command.

`+newgrp` [`–l`] [`––`] [*group*]

Equivalent to `exec /usr/bin/newgrp` ... *group*. `newgrp` makes the number of the specified group the current group number and overlaps itself with a shell.

`–l` `newgrp` overlaps the current shell with a login shell.

`––` must be specified if *group* begins with `–`.

`print`[`–Rnprsu`[*fildes*]] [*arg* ...]

`print` is the output mechanism of the Korn shell. If it is invoked with no options or with the option `–` or `––`, the arguments are printed on standard output as explained in the `echo` command description.

`–R` or `–r`

In raw mode, the escape conventions of `echo` are ignored. The `–R` option prints all subsequent arguments and options except for the `–n` option.

`-n` No newline is added to the output.

`-p` The arguments are written to the pipe of the process spawned with `|&`, not to standard output.

`-s` The arguments are written to the history file, not to standard output.

`-u`*fildes*
Arguments are written to the single-digit file descriptor *fildes*, not to standard output.

`pwd` See `print -r - $PWD` or the `pwd(1)` command.

`read` [`-prs`] [`-u` [*fildes*]] [`--`] [*name*`?`*prompt*] [*name* ...]
`read` is the input mechanism of the Korn shell. It reads one line from standard input and breaks it into fields, assigning the first field to the first *name*, the second field to the second *name*, etc., with leftover fields being assigned to the last *name*. If *name* is omitted, `REPLY` is used as the default *name*.

If the first argument contains a question mark, the remainder of this word is used as a prompt on standard error when the shell is interactive. The exit status is 0 unless an end-of-file is encountered.

`-r` The backslash `\` must not be treated as a special character.

`-p` The input line is read from the pipe of the process spawned with `|&`, not from standard input. An end-of-file from the pipe causes cleanup for this process so that a new process can be spawned with `|&`.

`-s` Saves the input as a command in the history file.

`-u` Input is read from the single-digit file descriptor *fildes*, not from standard input. The file descriptor can be opened with the built-in `exec` command. The default value of *fildes* is 0.

`--` If *name* begins with a minus sign (`-`), the end of the command line options must be marked with `--`.

`++readonly` [*name*=*value*] [*name* ...]

`readonly` [`-p`]

The variables specified by *name* are marked readonly. The values of these variables cannot be changed by subsequent assignment.

If you call `readonly` without any arguments, all read-only variables are written to the standard output. `-p` generates a portable format with read-only statements, that can be used to back up and restore a particular status.

`+return` [*num*]
Causes a shell function to return to the invoking script. The return status is *num* or that of the last command executed. If `return` is invoked while not in a function or a script executed with the dot command, it has the same effect as an invocation of `exit`.

`set` [*option* ...] [*arg* ...]
The `set` command can be used for three purposes:

- to set positional parameters
- to list variables on standard output
- to set Korn shell options

If the `-A` option is not specified, the positional parameters are assigned new values from the arguments in *arg*. The first argument is assigned to `$1`, the second to `$2`, etc.

If neither *option* nor *arg* is given, the names and values of all variables are listed on standard output.

The Korn shell options may be changed during a session. A minus sign before the option letters turns on the corresponding options; a plus sign turns them off. The current set of options may be found in the variable `$-`. These options can also be used upon invocation of the Korn shell. The available options are as follows:

`-A` *name*
Array assignment: unsets the variable *name* and assigns new values sequentially from the list of

arguments given in *arg*. If `+A` is used, the variable *name* is not unset first.

`-a` All subsequent variables that are defined are automatically exported.

`-b` The user is informed of all completed background processes.

`-C` If the output is redirected with >, no existing files are overwritten.

`-e` If a command has a non-zero exit status, the `ERR` trap, if set, is executed and the Korn shell exits.

`-f` Disables file name generation.

`-h` Each command executed is placed in the alias list as a tracked alias.

`-k` All variable assignment arguments are placed in the environment for a command, not just those that precede the command name.

`-m` Background jobs are run in a separate process group, and a report line is printed upon completion. The exit status of background jobs is reported in a completion message. This option is turned on automatically for an interactive Korn shell.

`-n` Reads commands and check them for syntax errors, but does not execute them. This option is ignored by an interactive Korn shell.

`-o` *argument*

argument can be one of the following option names:

`allexport`
Same as `-a`.

`bgnice`
All background jobs are run at a lower priority. This is the default mode.

`emacs` The commands are edited in `emacs` mode.

`errexit`
Same as `-e`.

`gmacs` As for `emacs`. See the section **emacs editing mode** above.

`ignoreeof`
The Korn shell will not exit on encountering an end-of-file. The built-in `exit` command must be used.

`interactive`
Interactive Korn shell.

`keywords`
Same as `-k`.

`markdirs`
All directory names resulting from file name generation have a trailing slash appended.

`monitor`
Same as `-m`.

`noclobber`
Prevents output redirection with `>` from overwriting existing files. Corresponds to `-C`.

`noexec`
Same as `-n`.

`noglob`
Same as `-f`.

`nolog` Function definitions are not saved in the history file.

`notify`
Corresponds to `-b`.

`nounset`
Same as `-u`.

`privileged`
Same as `-p`.

`restricted`
Restricted Korn shell. See chapter **rksh only** above.

`trackall`
Same as `-h`.

`verbose`
Same as `-v`.

`vi` Puts you in the insert mode of a `vi` style in-line editor until you press <ESCAPE>. This puts you in command mode. A newline character sends the line.

`viraw` In `vi` mode, each character is processed as soon as it is typed.

`xtrace`
Same as `-x`.

`-p` If the effective user ID (group ID) is not equal to the real UID (GID), processing of `$HOME/.profile` is disabled and the file `/etc/suid_profile` is used instead of the `ENV` file. Turning this option off with `+p` causes the effective UID and GID to be set to the real UID and GID.

`-s` Sorts the positional parameters lexicographically.

Note that this option is not the same as the `-s` option that can be used when invoking the Korn shell.

`-t` Terminates the Korn shell after reading and executing one command.

`-u` Unset parameters are treated as errors when substitution is performed.

`-v` The Korn shell prints input lines as it reads them.

`-x` Commands and their arguments are printed as they are executed.

`-` Turns off the `-x` and `-v` options and stops examining arguments for options. The rest of the command line is taken as *arg*.

`--` Stops any of the options being changed; useful in setting `$1` to a value beginning with `-`. If no arguments follow this option, the positional parameters are unset.

`+shift` [*number*]
The positional parameters from `$`*number*+1 onward are renamed `$1` and so on, i.e. the values of the positional parameters are shifted to the left. The default value for *number* is 1. `$0` is not affected by this command. The *number* argument can be any arithmetic expression that evaluates to a positive integer less than or equal to `$#`.

`test` See **Conditional expressions** above or the `test(1)` command.

`+times`
Prints the accumulated user and system times for the current Korn shell and for processes run from the shell.

`+trap` [*cmd*] [*sig* ...]
cmd is a command or list of commands to be read and executed when the Korn shell receives signal(s) *sig*.

(Note that *cmd* is scanned twice, once when the shell executes the `trap` command and once when *cmd* is executed in response to a signal.)

Each *sig* can be given as a number or as the name of the signal (see `kill`).

`trap` commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on invocation of the current shell is ineffective. If *cmd* is omitted or is in the form of a minus sign (-), all trap(s) on *sig* are reset to their original values. If *cmd* is the null string (""), the associated signal is ignored by the Korn shell and by the commands it invokes. If *sig* is `ERR`, *cmd* will be executed

whenever a command has a non-zero exit status. If *sig* is `DEBUG`, *cmd* will be executed after each command. If *sig* is `0` or `EXIT` and the `trap` statement is executed inside the body of a function, *cmd* is executed after the function completes. If *sig* is `0` or `EXIT` for a trap set outside any function, *cmd* is executed on termination of the shell.

The `trap` command with no arguments writes a list of commands associated with each signal number to standard output.

`true` See `:` or the `true(1)` command.

`++typeset` [*option* ...] [*name*[=*value*]]...

The `typeset` command can be used for two purposes:

- to list variable names and other optional information on standard output,
- to set attributes and values for shell variables.

If neither *option* nor *name*[=*value*] is given, the names and attributes of all variables are listed on standard output.

If there are options but no *name* arguments, all variables which have these options set are listed along with their attributes.

Using a plus sign instead of a minus sign stops the values from being printed.

option can be used to assign one or more attributes to the variable *name*[=*value*]. When `typeset` is invoked inside a function, a new instance of the variable *name* is created. The value and its attribute are restored when the function completes.

Using a minus sign before the option letters turns on the corresponding attributes; a plus sign turns them off.

The following attributes may be specified:

`-H` This option allows for UNIX-style mapping of file names to host names on non-UNIX machines.

`-L`[*num*]

Left-justifies *value* and removes leading blanks.

If *num* is non-zero, it defines the length of the field; otherwise, the field length is determined by the length of the first value assignment.

This length is used for subsequent assignments. When the variable is reassigned, it is padded to the right with blanks or truncated as necessary.

Leading zeros are removed if the `-Z` option is also set. The `-L` option turns off the `-R` option.

`-R`[*num*]

Right-justifies *value* and pads it with leading blanks.

If *num* is non-zero, it defines the length of the field; otherwise, the field length is determined by the length of the first value assignment. This length is used for subsequent assignments.

When the variable is reassigned, the field is either padded to the left with blanks or truncated from the end.

The `-R` option turns off the `-L` option.

`-Z`[*num*]

Right-justifies the *value* and pads it with leading zeros if the first non-blank character is a digit and the `-L` option has not been set. If *num* is non-zero, it defines the length of the field; otherwise, the field length is determined by the length of the first value assignment.

`-f` The names refer to function names rather than variable names. The Korn shell stores the functions in the `.sh_history` file. Consequently, you cannot display a function definition on the screen if `.sh_history` does not exist or if the `nolog` option was not set when the function was read. No assignments can be made, and the only other valid options in combination with `-f` are:

`-t` Turns on execution tracing for the function.

-u Causes the function to be marked undefined. The FPATH variable is searched to find the function definition the next time the function is referenced.

-x Allows the function definition to remain in effect across Korn shell scripts invoked by name. In other words, the function is exported.

-i[*num*]
The variable is an integer. This option makes arithmetic faster. If *num* is non-zero, it defines the output arithmetic base; otherwise, the first assignment determines the output base.

-l All uppercase characters are converted to lowercase. The -u option is turned off.

-r The given variables are marked readonly, which means that their values cannot be changed by subsequent assignment.

-t Tags the variables. Tags are user-definable and have no special meaning to the Korn shell.

-u All lowercase characters are converted to uppercase characters. The -l option is turned off.

-x The given names are marked for automatic export to any new environment.

ulimit [*option* ...]

ulimit [*option* ...] *limit*

ulimit can be used to list or set resource limits for your processes. These limits are described in getrlimit(2) .

Format 1: List resource limits

ulimit lists the limits specified by *option* (see below) on standard output. The options may be combined in any order. The -a option can be used to list all limits.

Format 2: Set resource limits

ulimit sets the limit specified by the option to *limit*. The string *unlimited* requests the largest valid limit. Limits may be set for only one resource at a time. Any user may set a soft limit to any value below the hard limit. Any user may lower a hard limit, but only the system administrator may raise a hard limit [see su(1)].

The -H option specifies a hard limit. The -S option specifies a soft limit. If neither option is specified, ulimit will set both limits and list the soft limit.

The following options specify the resource whose limits are to be listed or set. If no option is given, -f is assumed.

-c maximum core file size (in 512-byte blocks)

-d maximum size of data segment or heap (in Kbytes)

-f maximum file size (in 512-byte blocks)

-n maximum number of file descriptors plus 1

-s maximum size of stack segment (in Kbytes)

-t maximum CPU time (in seconds)

-v maximum size of virtual memory (in Kbytes) .RE

umask [-S] [--] [*mask*]
umask sets or displays the current user file-creation mode mask. If *mask* is omitted, the current value of the mask is written to standard output. If a *mask* is specified, the mode mask is set accordingly [see umask(1)]. *mask* can be either an octal number or a symbolic value as described in chmod(1) . If a symbolic value is used, the new umask value is the complement of the result of applying *mask* to the complement of the previous umask value.

-S Symbolic output in the format u=*rights_user*, g=*rights_group*, o=*rights_other*, whereby *rights_** is output as a combination of the letters r, w, x.

-- If *mask* begins with a minus sign (-), the end of the command line options must be marked with --.

`unalias` [−a] [*name* ...]
: The named variables are removed from the alias list. If −a is specified, all alias definitions are removed from the alias table.

`unset` [−fv] *name* ...
: The named variables are unassigned, i.e. their values and attributes are erased. Read-only variables cannot be unset. Unsetting ERRNO, LINENO, MAILCHECK, OPTARG, OPTIND, RANDOM, SECONDS, TMOUT and _ removes their special meaning even if they are subsequently assigned to. If the −f option is set, then the names refer to function names, while in the case of −v they refer to the names of variables.

`+wait` [*job* ...]
: Waits for the specified job to terminate and report its exit status. If *job* is not given, all currently active child processes are waited for. The exit status of `wait` is that of the process waited for. The format of *job* is as described under **Jobs** above.

`whence` [−pv] *name* ...
: Indicates how each name would be interpreted if used as a command name.

  −v produces a more detailed report.

  −p does a path search for *name* even if *name* is an alias, a function, or a reserved word.

EXIT STATUS

The Korn shell normally returns the exit status of the last command executed [see `exit(1)`].

`1-125` Errors detected by the Korn shell, such as syntax errors, cause it to return a non-zero exit status. If the Korn shell is being used non-interactively, execution of the shell script is aborted. Runtime errors detected by the shell are reported by printing the command or function name and the error condition. If the line number that the error occurred on is greater than one, the line number is also printed in square brackets (`[...]`) after the command or function name.

`126` The specified command exists, but could not be invoked.

`127` The specified command could not be found by a non-interactive shell.

FURTHER DIFFERENCES BETWEEN KORN SHELL – BOURNE SHELL

Since the Korn shell (`ksh`) and Bourne shell (`sh`) do not always behave in the same way, the shell scripts also behave differently (see also chapter **Major Korn shell features not available in the Bourne shell** above).

The following differences are known to exist at present:

1. If an `exit` is executed in a loop (`for`, `until`, `while`) and if the input is read from a file, the Bourne shell quits the loop and the Korn shell ends the script. Example of a `while` loop:

```
while read i
do
        echo $i
        if [ "$i" = "end" ]
        then
              exit 199
        fi
done < inputfile
retflag=$?
echo $retflag
```

2. The Shell `cd` built-in behaves differently when metacharacters (*, ?) are used.

   The current directory contains the subdirectories `dir_cmd`, `dir_lib` and `dir_head`.

   `cd dir_*`

   The Korn shell generates the error message "bad argument count", and the Bourne shell changes to

the directory with the first pattern matching (`dir_cmd`).

3. Call parameters are interpreted differently.

   Invoking at the shell level:

   ```
   ((/tmp/shelltest add "-po XON") 2>&1)
   ```

   Korn shell:

   $1 = `add` $2 = `-po` $1 = `XON`

   Bourne shell:

   $1 = `add` $2 = `-po XON`

4. In the Korn shell, `time` is a keyword at the start of a line (like `if`, `then` etc.; cf. above, section **Reserved words**). In the Bourne shell, on the other hand, `time` is a standard user command (`/usr/bin`). The following deviations regarding `time` can be distinguished.

   The behavior of the

   `time` *pipeline*

   command varies. In the case of the Korn shell, the pipeline commands are executed and then the elapsed time, the user and system time for these commands are reported as statistics to standard error.

   With the Bourne shell, `time` is only executed for the first pipeline command and the statistics are output beforehand.

   Thus in the case of the Korn shell, if you issue the

   ```
   time ls | pg
   ```

   call, you will first see the output of `ls` and the `pg` input must then be performed. It is only when everything has ended that the statistics are output for `ls` and `pg`.

   With the Bourne shell, the statistics are output first, followed by the output of `ls` and the input for `pg` [cf. xpg4_sh(1)].

5. The Korn shell supplies the function name for a shell script function in `$0` while the Bourne shell also supplies the name of the shell script.

   For example, in the following shell script called `test.sh`:

   ```
   dummy()□
   {□
       echo $0□
   }□
   □
   dummy
   ```

   The Korn shell echoes `dummy`, the Bourne shell echoes `test.sh` [cf. xpg4_sh(1)].

6. If the Bourne shell discovers an error when executing the `/etc/profile` file, it issues an error message and immediately discontinues execution of the file. The Korn shell, on the other hand, issues an error message, but then continues executing – if this is possible – the remainder of the `/etc/profile` file. Example of a `/etc/profile` file:

   ```
   echo "previously"□
   echo $TERM > /.term.$$          # no write access□
   echo "afterwards"
   ```

   The Korn shell executes the "afterwards" line, while the Bourne shell does not.

FILES

`/etc/passwd`
`/etc/profile`
`/etc/suid_profile`

```
$HOME/.profile
/tmp/sh*
/dev/null
```

SEE ALSO

cat(1), cd(1), chmod(1), colon(1), cut(1), dot(1), echo(1), env(1), newgrp(1), paste(1), stty(1), test(1), umask(1), vi(1), xpg4_sh(1), dup(2), exec(2), fork(2), getrlimit(2), ioctl(2), lseek(2), pipe(2), signal(2), ulimit(2), umask(2), wait(2), rand(3C), a.out(4), profile(4), environ(5), signal(5). □

M. I. Bolski and D. G. Korn, *The Korn-Shell Command and Programming Language*

last(1) - ls(1-ucb)

User's Reference Manual

last(1)

NAME

`last -` indicate last user or terminal login

SYNOPSIS

`last` [–[`n`] *number*] [–`f` *filename*] [*name ...*]

DESCRIPTION

`last` can be used to list the times when users have logged in and out.

OPTIONS

No option specified:
: `last` lists all the login and logout times recorded in `/var/adm/wtmpx` and issues a message of the form "`wtmpx begins`*date*", indicating the date of the oldest entry.

[`n`] *number*
: Limit the number of login and logout times displayed to *number*.

–*number*
–`n` *number*
: Limit the number of login and logout times displayed to *number*, but without printing the "`wtmpx begins` *date*" message.

–`f` *filename*
: Specifies *filename* as the accounting file from which login and logout times are read. The default accounting file is `/var/adm/wtmpx`.

name One or more login names and/or terminal numbers (`tty???`).

Mode of operation

The file `/var/adm/wtmpx` maintains a record of all login and logout times. `last` retrieves all the information about a user and the associated terminals from this file.

The arguments to `last` specify the login names or terminals for which information is to be retrieved. The names of terminals must be given in full.

`last` displays the following information:

- the login name
- the terminal
- where appropriate, the name of the host from which the user logged in
- the login and logout times

By default, `last` displays a record of all logins and logouts in reverse order. If specific users or terminals are supplied as arguments, only the information for the selected users or terminals will be output. If multiple arguments are given, the information which applies to each individual argument is printed.

`last` also indicates if the session is still continuing or was cut short by a `reboot`. The pseudo user `reboot` logs in each time the system is rebooted.

If `last` is interrupted, it prints a message on the standard output to indicate how far the search had progressed until the time of the interrupt. If `last` is interrupted with `SIGQUIT` (CTRL-\), however, it will write the same message to standard output and then continue the search.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

User Jon wishes to determine the last three times at which he logged in at terminal `tty006`. He also wishes to know if other users have been working at the same terminal.

```
$ last –3 Jon tty006
Jon   tty006              Fri  Aug   3   13:37 - 15:53  (02:15)
Jon   tty006              Thu  Aug   2   10:25 - 12:56  (02:31)
Jon   tty006              Wed  Aug   1   09:54 - 11:47  (01:53)
```

FILES

`/var/adm/wtmp`
`/var/adm/wtmpx`

The file `/var/adm/wtmp`[x] contains information about users and accounting information. The login and logout times are recorded in this file.

SEE ALSO

`utmp`(4) .

User's Reference Manual

lastcomm(1)
BSD Compatibility Package

NAME

`lastcomm` - show the last commands executed, in reverse order

SYNOPSIS

`/usr/ucb/lastcomm` [*command-name* ...] [*user-name* ...] [*terminal-name* ...]

DESCRIPTION

The `lastcomm` command gives information on previously executed commands. `lastcomm` with no arguments displays information about all the commands recorded during the current accounting file's lifetime. If called with arguments, `lastcomm` only displays accounting entries with a matching *command-name*, *user-name*, or *terminal-name*.

EXAMPLES

The command:

```
lastcomm a.out root term/01
```

would produce a listing of all the executions of commands named `a.out`, by user `root` while using the terminal `term/01` and

```
lastcomm root
```

would produce a listing of all the commands executed by user `root`.

For each process entry, `lastcomm` displays the following items of information:

- The command name under which the process was called.
- One or more flags indicating special information about the process. The flags have the following meanings:
  - `F` The process performed a `fork` but not an `exec`.
  - `S` The process ran as a set-user-id program.
- The name of the user who ran the process.
- The terminal which the user was logged in on at the time (if applicable).
- The amount of CPU time used by the process (in seconds).
- The date and time the process exited.

FILES

`/var/adm/pacct`
accounting file

SEE ALSO

`last`(1), `sigvec`(3), `acct`(4).

less(1)

NAME

`less -` control the output on the screen

SYNOPSIS

`less` [–[+]aBcCdeEfFHiIlLmMqQrsuUw] [–b*number*] [–j*number*] [–[n]*number*] [–k*number*] [–I*number*] [–P[mM=]*string*] [–o *file* | –O *file*] [–K*file*] [–t*tag*] [–T*tagsfile*] [–p*command*] [*file*...]

`less –h`

DESCRIPTION

`less` is identical to `more`. See `more(1)` .

User's Reference Manual

lesskey(1)

NAME

`lesskey` - define key bindings for less or more

SYNOPSIS

`lesskey` [`–o` *outputfile*] [*inputfile*]

DESCRIPTION

`lesskey` is used to specify key bindings which should be used by `less` or `more`. *inputfile* is a text file which describes the key bindings. *outputfile* is a binary file which is used by `less` or `more`. If no *inputfile* is specified, the default input is used. If no *outputfile* is specified, the file `$HOME/.less` is created.

The *inputfile* file consists of lines of the format:

string <*space*> *action* [*extra-string*] <*newline*>

<*space*> is any sequence of one or more spaces and/or tabs. *string* describes the command key(s) which invoke the action. *string* can be a single command key or a sequence of up to 15 keys. *action* is the name of the action to be executed by `less`. The names are listed below. *string* can contain normal characters and control characters. Control characters begin with a carat. A backslash means that the following character is regarded as a normal character. The carat, space and tab as well as the backslash must be preceded by a backslash. A backslash followed by one to three octal digits can be used to specify a character by its octal value. Blank lines and lines which start with a # sign are ignored.

An *action* can be followed by an *extra-string*. *extra-string* is parsed after the command is entered, as if it were an entry in the command line. This can be useful to extend the functionality of a command. For example, the following entries create a pair of commands to turn on/off line numbering using the `vi` synopsis:

```
:set\ nu            toggle-option –N
:set\ nonu          toggle-option +N
```

ACTIONS

In *action* you specify the function of `more` or `less` which is to be initiated by the command key(s). `lesskey` recognizes the following keywords (each with an explanation):

`back-bracket`	search bracket backward
`back-line`	back a line
`back-line-force`	back a forced line
`back-screen`	back a screen
`back-scroll`	scroll backward

`back-search`	search backward
`back-window`	back a window
`debug`	activate debug mode
`display-flag`	like `display-option`
`display-option`	display value of an option
`end`	quit `less` or `more`
`examine`	display file
`first-cmd`	start command
`firstcmd`	start command

`flush-repaint`	refresh screen, delete buffer
`forw-bracket`	search bracket forward
`forw-forever`	always forward
`forw-line`	forward a line
`forw-line-quit`	line forward, at end of line continue to next file or quit `less` or `more`
`forw-line-skip`	skip a line
`forw-line-force`	forward a forced line
`forw-screen`	forward a screen
`forw-scroll`	scroll forward
`forw-search`	search forward
`forw-window`	forward a window
`goto-end`	go to end
`goto-line`	go to a line
`goto-mark`	go to a mark
`help`	help
`index-file`	file from a list
`invalid`	invalid
`next-file`	next file
`noaction`	(no action. Useful to turn off commands silently)
`percent`	go to percent
`pipe`	pipe
`prev-file`	previous file
`quit`	quit
`repaint`	refresh screen
`repaint-flush`	refresh screen, delete buffer

`repeat-search`	repeat search
`repeat-search-all`	repeat search across all files
`reverse-search`	repeat search backward
`reverse-search-all`	repeat search backward across all files
`set-mark`	set mark
`shell`	shell command
`status`	status
`toggle-flag`	like `toggle-option`
`toggle-option`	reverse option
`version`	display version
`visual`	call editor

Commands specified by `lesskey` have priority over the default commands. A default command can be

deactivated by assigning it the action "invalid" in the key file.

EXAMPLE

The following input file describes default command keys used by `less`:

```
e       forw-line
j       forw-line
^E      forw-line
^N      forw-line
k       back-line
y       back-line
^Y      back-line
^K      back-line
^P      back-line
J       forw-line-force
K       back-line-force
Y       back-line-force
d       forw-scroll
^D      forw-scroll
u       back-scroll
^U      back-scroll
\40     forw-screen
f       forw-screen
^F      forw-screen
^V      forw-screen
b       back-screen
^B      back-screen
\33v    back-screen
z       forw-window
w       back-window
F       forw-forever
R       repaint-flush
r       repaint
^R      repaint
^L      repaint
g       goto-line
<       goto-line
\33<    goto-line
p       percent
%       percent
{       forw-bracket {}
}       back-bracket {}
```

```
(       forw-bracket ()
)       back-bracket ()
[       forw-bracket []
]       back-bracket []
\33^F   forw-bracket
\33^B   back-bracket
G       goto-end
\33>    goto-end
>       goto-end
P       goto-end
=       status
^G      status
:f      status
/       forw-search
?       back-search
\33/    forw-search *
\33?    back-search *
n       repeat-search
\33n    repeat-search all
N       reverse-search
\33N    reverse-search all
m       set-mark
'       goto-mark
^X^X    goto-mark
E       examine
:e      examine
^X^V    examine
:n      next-file
:p      prev-file
:x      index-file
_       toggle-option
:t      toggle-option t
s       toggle-option o
_       display-option
|       pipe
v       visual
!       shell
+       firstcmd
H       help
h       help
```

```
V       version
q       quit
:q      quit
:Q      quit
ZZ      quit
\33\33  quit
```

SEE ALSO

less(1), more(1).

User's Reference Manual

lfmt(1)

NAME

lfmt - display error message in standard format and pass to logging and monitoring services

SYNOPSIS

lfmt [-c] [-f *flags*] [-l *label*] [-s *severity*] [-g *catalog*:*msgnum*] *format* [*args*]

DESCRIPTION

lfmt uses *format* for printf style formatting of *args*. If the -g option is specified, lfmt retrieves a localized version of the *format* string from a locale-specific message database. The output is displayed on standard error (stderr).

lfmt encapsulates the output in the standard error message format.

lfmt forwards its output to the logging and monitoring facility. Optionally, lfmt will display the output on the console, with a date and time stamp.

OPTIONS

-c The -c option causes the message to also be displayed along with a date/time stamp on the console.

-f The -f *flags* option specifies logging information as a comma-separated list of keywords from the sets:

Major classification

Identifies the source of the condition. Identifiers are:

```
hard (hardware),
soft (software),
firm (firmware).
```

Message source subclassification

Identifies the type of software in which the problem is spotted. Identifiers are:

```
appl (application),
util (utility),
opsys (operating system).
```

-l The -l *label* option specifies the label string to be displayed with the message (e.g. "UX:cat").

label is a character string no more than 25 characters in length; it will be automatically suffixed with a colon (:). When unspecified, no label is displayed as part of the message.

-s The -s *severity* option specifies the severity string to be displayed with the message. Acceptable strings include the standard severities in either their print string (i.e. HALT, ERROR, INFO, WARNING, and "TO FIX") or keyword (i.e. halt, error, info, warn, and action) forms, or any other user-defined string. A user-defined string will be assigned the integer severity value of 5. The severity will be suffixed with a colon (:). The ERROR severity will be used if no severity is specified.

-g The -g *catalog*:*msgnum* option specifies that a localized version of the *format* should be retrieved from a message database.

catalog is used to indicate the message database that contains the localized version of the *format* string. *catalog* must be limited to 14 characters. These characters must be selected from a set of all characters values, excluding \0 (null) and the ASCII codes for / (slash) and : (colon).

msgnum is a positive number that indicates the index of the string into the message database.

If the catalog does not exist in the current locale (identified by the LC_MESSAGES or LANG environment variables), or if the message number is out of bounds, lfmt will attempt to retrieve the message from the C locale. If this second retrieval fails, lfmt uses the *format* string as passed on the command line.

lfmt will output Message not found!!\n as the *format* string if *catalog* is not a valid catalog name, or if *msgnum* is not a valid number.

STANDARD ERROR MESSAGE FORMAT

`lfmt` displays error messages in the following format:

label`:` *severity*`:` *text*

If no *label* was defined using the `-l` *label* option, the message is displayed in the format:

severity`:` *text*

The environment variable `LC_ALL_LABELS` is used to determine if labels will be added to the text of the messages. The value of `LC_ALL_LABELS` is consulted during the formation of the first message. Labels will be added (or not added) to all subsequent messages in the same way. Changing or setting `LC_ALL_LABELS` after the first message will have no effect message formation within the same execution.

If `LC_ALL_LABELS` is set to the value 1, the label string will be included with the message. If `LC_ALL_LABELS` does not exist, or is empty, or is set to 0, only the text of the message is displayed. Any other values of `LC_ALL_LABELS` will produce undefined results.

For example, consider the following message produced when `LC_ALL_LABELS` is set to 1:

```
UX:cat: ERROR: Cannot open door: No such file or directory
```

If `LC_ALL_LABELS` had not existed, or was empty, or had been set to 0, the above message would have been:

```
cat: Cannot open door: No such file or directory
```

If `lfmt` is called twice to display an error message and a helpful action or recovery message, the output can look like:

label`:` *severity*`:` *text*
label`:` `TO FIX:` *text*

EXIT STATUS

Upon success, `lfmt` exits with code 0. Upon failure, `lfmt` exits with the following codes:

`1` write error.

`2` cannot log or forward to console.

`3` syntax error.

EXAMPLES

```
lfmt -fsoft,util -l UX:test -s info "test facility enabled\n"
```

displays the message to stderr and makes it available for logging:

```
UX:test: INFO: test facility enabled
```

SEE ALSO

`fmtmsg(1)`, `gettxt(1)`, `pfmt(1)`, `printf(1)`, `environ(5)`.

User's Reference Manual

line(1)

NAME

`line` - read one line

SYNOPSIS

```
line
```

DESCRIPTION

`line` copies one line from the standard input (including the newline character) and writes it on the standard output. The minimum output is one newline character. The `line` command is typically used in shell scripts when user input is to be processed interactively.

You can also use the built-in command `read` instead of `line`.

EXIT STATUS

0 normal exit

>0 end-of-file character (EOF) encountered in input

EXAMPLES

The shell script *customers* is to be used to check whether a particular customer is entered in the file named `list`.

Contents of *customers*:

```
echo "Enter a customer name and hit return."
NAME=`line`
if grep $NAME list
then echo "Customer $NAME is already in list"
else echo "Customer $NAME is not in list"
fi
```

When the script *customers* is run (execute permission is required), the user is prompted to enter a customer name. The input is then read in by `line` and assigned as the value of the variable `NAME`. The `if` statement checks the exit status of the `grep` command. If this value is 0, i.e. `grep` was successful in locating the value of the `NAME` variable (`$NAME`) in the file `list`, the message `Customer $NAME is already in list` is displayed.

A non-zero exit status means that `grep` has not found the name, in which case the message `Customer $NAME is not in list` is issued.

SEE ALSO

`read`(1), `sh`(1), `read`(2).

listusers(1)

NAME

`listusers -` list user login information

SYNOPSIS

`listusers` [*option ...*]

DESCRIPTION

`listusers` prints information on user logins entered in the `/etc/passwd` file. A user login is defined as a login name that has a user ID (UID) of 100 or greater. A user group is defined as a group with a group ID (GID) of 100 or greater.

OPTIONS

No option specified:

`listusers` displays the following information:

- alist of all user logins entered in the `/etc/passwd` file,
- any associated entries in the comment field of the `/etc/passwd` file.

–g*group*[,*group ...*]

Lists all user logins belonging to the specified user *group*. Multiple groups can be specified as a comma-separated list. Users who belong to more than one of the selected groups are only listed once.

–l*login*[,*login ...*]

Lists the user login specified by *login* if that login name is entered in the `/etc/passwd` file. If there is an entry associated with the login in the comment field of the `/etc/passwd` file, that entry is also displayed. Multiple logins may be specified as a comma-separated list.

ERROR MESSAGES

`[group] is not a user group`

With the `–g` option you have specified a group with a GID below 100.

`[group] was not found`

With the `–g` option you have specified an illegal group or listed multiple groups without separating them with commas.

`[login] is not a user login`

With the `–l` option you have specified a login with a UID below 100.

`[login] was not found`

With the `–l` option you have specified an illegal login or listed multiple logins without separating them with commas.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

All login names entered in the `/etc/passwd` file are to be listed. Associated entries in the comment field of `/etc/passwd` are shown in the second column.

```
$ listusers
carstens     Cons. 1.4. to 30.09., Tel. 43196
cmd          Command project '93
guest
install      Initial login
martin
lab24
nobody       non-privileged user
andrew
vmsys        FACE executables
```

Example 2

All users belonging to the groups `consult` and `usrother` are to be listed. The user with the login name `carstens` belongs to both groups, but is only listed once.

```
$ listusers –gconsult,usrother
carstens     Cons. 1.4. to 30.09., Tel. 43196
guest
lab24
```

FILES

`/etc/group`
: File containing group names

`/etc/passwd`
: File containing login names

User's Reference Manual

ln(1)

NAME

`ln -` make links to files

SYNOPSIS

Format 1

`ln` [*option*] *source target*

Format 2

`ln` [*option*] *source ... targdir*

Format 3

`ln -s` [`--`] *source target*

Format 4

`ln -s` [`--`] *source ... targdir*

DESCRIPTION

`ln` creates links to existing files or directories. This allows you to access these files and/or directories under various file names or path names (see **FUNCTIONALITY**).

To make a link with `ln`, you need write permission for the directory in which the link is to be entered.

There are two types of link:

- Hard link:

  When you create a hard link to a file, there are multiple directory entries for this file, either in one directory or in different directories, but the physical file is only present once. The inode for each file contains a link counter which is decremented by one when a link is removed. The file itself is not deleted until all links to it have been removed.

  A hard link cannot be used to refer to a directory or to files on different file systems.

- Symbolic link:

  A symbolic link is a file that contains a path name. When the shell encounters a file name that represents a symbolic link, it replaces this name with the specified path name. Thus what you access is not the symbolic link, but the file to which the path name points.

  You can set up symbolic links to any files or directories, even to those on different file systems.

OPTIONS

Format 1: Make a hard link

`ln` [*option*] *source target*

`ln` makes a new link named *target* to the file named *source*. The file can then be accessed under both names, i.e. *source* and *target*.

No option specified:
: If a file with the same name as that to be assigned to the link to be created already exists, `ln` returns an error message. The existing file is not overwritten.

  This corresponds to the old option `-n`, which will continue to be supported for compatibility reasons.

`-f` (force) If a file named *target* already exists, `ln` creates the link without asking questions. If *link* was the only link to the original file, the contents of the file are deleted after `ln` is executed.

`--` If *source* begins with a dash (-), the end of the command-line options must be marked with `--`.

source
: Name of the file to which you want to make a link. The file must already exist when you call `ln`. You are

not allowed to specify a directory as the *source*.

target

Name of the link that you wish to make to *source*. *target* can be a simple file name (basename) or an absolute or relative path name:

a basename:

`ln` enters the basename *target* in the current working directory.

an absolute or relative path name in the form *prefix*/*name*:

`ln` enters the basename *name* in the directory identified as *prefix*.

If the parent directory of *target* is writable but has the sticky bit set, one of the following conditions must be fulfilled in order to for *target* to be created:

- the file must be owned by the user.
- the directory must be owned by the user.
- the user must have write permission for the file.
- the user must be a privileged user.

This format of `ln` cannot be used to create links that span different file systems. Such links can be made using the `–s` option (see **Format 3** and **Format 4**).

Format 2: Make a hard link with the same name in another directory

`ln` [*option*] *source ... targdir*

`ln` creates a link to the file (or files) named *source* in another directory named *targdir*. The file can then be accessed in two different directories under the same basic file name (basename).

No option specified:

If a file with the same name as that to be assigned to the link to be created already exists, `ln` returns an error message. The existing file is not overwritten.

This corresponds to the old option `–n`, which will continue to be supported for compatibility reasons.

`–f` (force) If *targdir* already contains a file named *source*, `ln` creates the link without asking questions, If *link* was the only reference to the original file, the contents of the file are deleted after `ln` is executed.

`––` If *source* begins with a dash (–), the end of the command-line options must be marked with `––`.

source

Name of the file to which you want to make a link. The file must already exist when you call `ln`. You are not allowed to make links to directories. You can name any number of files in one call. The specified *source* can be a file basename or an absolute or relative path name:

a basename:

`ln` enters the basename *source* in the named *targdir*.

an absolute or relative path name in the form *prefix*/*name*:

`ln` enters the basename *name* in the named `targdir`.

targdir

Name of the directory in which the link is to be entered. The directory must already exist.

This format of `ln` cannot be used to create links that span different file systems.

Format 3: Make a symbolic link

`ln –s` [`––`] *source target*

`ln –s` makes a symbolic link named *target* to the named *source*, where *source* can be a file or a directory. The main difference with `ln –s` is that it creates symbolic links, which can span different file systems. This is not possible with hard links.

`––` If *source* begins with a dash (–), the end of the command-line options must be marked with `––`.

source

Name of the file or directory to which a symbolic link is to be made. *source* may be specified as any path name and need not exist. It may also reside on a different file system from *target*.

target

Name of the symbolic link that you wish to make to *source*. *target* can be a file basename or an absolute or relative path name:

a basename:

`ln` enters the basename *target* as a symbolic link in the current working directory.

an absolute or relative path name in the form *prefix*/*name*:

`ln` enters the basename *name* in the directory identified as *prefix*.

If a file named *target* already exists, an error message is returned (see **ERROR MESSAGES**). The existing file is not overwritten.

Format 4: Make a symbolic link with the same name in another directory

`ln –s` [––] *source ... targdir*

For each file or directory that is specified as *source*, `ln` creates a symbolic link in the directory *targdir*. These links can span different file systems.

–– If *source* begins with a dash (–), the end of the command-line options must be marked with ––.

source

Name of the file or directory to which you want to make a symbolic link. You can name any number of sources in one call.

The specified *source* can be an absolute or relative path name:

an absolute or relative path name in the form *prefix*/*name*:

`ln` enters the basename *name* in the directory *targdir* as a symbolic link to *prefix*/*name*.

If *targdir* already contains a file with the same basename as *name*, an error is returned (see **ERROR MESSAGES**), and the existing file is not overwritten.

targdir

Name of the directory in which the symbolic links are to be created. This directory must exist.

FUNCTIONALITY

Hard links

When `ln` makes a link to a file, the basic file name (basename) associated with the link is entered in the appropriate directory. This entry receives the same inode number as the original file name. Thus, both file names have the same inode, and consequently the same attributes (access permissions, owner, dates, etc.). The physical file referenced by the file names is only present once, but the user can now access the same file under different file or path names (see *Example 1*).

The inode number indicates whether two file names are linked to the same file (see `ls –i`); the number of links shows you how many directory entries exist for the file (see `ls –l`).

The `rm` command can be used to remove an entry (link) from a directory. If there was more than one link to the file, it can still be accessed under the remaining names. `rm` does not delete the file itself until the last link (i.e. name) is removed.

Symbolic links

A symbolic link is a file that contains a path name. These path names can be listed with the command `ls –l`. When the shell encounters a file name that represents a symbolic link, it replaces this name with the specified path name. In other words, one path name is mapped to another. There is no link counting mechanism in this case; when a symbolic link is removed, the file containing the path name is deleted.

The inode of the physical file that is referenced by a symbolic link does not contain any information about the link. The link counter only keeps track of hard links. This means that if you delete the target of a symbolic link,

the link will continue to exist, but will now refer to a file having no contents and no inode. Symbolic links are not restricted by file system boundaries, and the path names contained in them may refer both to files and to directories.

Symbolic links can be shown with the `ls` command:

- `ls -l` indicates which files in the specified directory are symbolic links. The contents of each symbolic link, i.e. the path name of the referenced file, are shown following the file name and the symbol "->".
- `ls -L` provides information on the file or directory that is referenced by the symbolic link.

Operations involving ".." (such as "`cd ..`") in a directory that is symbolically linked will reference the original directory, not the target.

ERROR MESSAGES

`ln: no permission for` *link*

`ln` cannot create the named *link*, since you do not have write permission for the directory in which the link is to be entered.

`ln: cannot access` *file*

`ln` cannot access the named *file*, either because it does not exist or because you have no execute permission (x bit) for a directory that appears in the path name of *file*.

`ln: <`*dir*`> directory`

You have specified the directory *dir* instead of a file. `ln` only establishes hard links to regular files, special files, or FIFO files, not to directories.

`ln: different file system`

You have tried to create a link to a file and enter it in a different file system. `ln` does not create hard links across file systems; try using the `-s` option.

`ln: cannot create` *file*

`ln: File exists`

You have tried to create a link named *file* in a directory, but there is already a file under the same name in that directory.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

Answers to yes/no queries must be given in the language appropriate to the current locale.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

User Pat is the owner of the files `frogs` and `snails`, both of which are in the directory `/home/pat`. Mary is to now work with these files as well. To continue working in her own directory and save herself the bother of entering long path names, Mary establishes links to these files in her own directory `/home/mary`. To be able to do this, she has to have:

- write permission for her own directory `/home/mary`
- execute permission for Pat's directory `/home/pat`:

To be able to read and modify the files, Mary also needs read and write permission for the files themselves. In our example these conditions are met if Mary is a member of the group `proj` (see below). The fastest way for Mary to set up the links is to call `ln` in **Format 2**:

```
$ ln /home/pat/frogs /home/pat/snails /home/mary
```

The same result can be achieved by calling `ln` twice in **Format 1**:

```
$ ln /home/pat/frogs /home/mary/frogs
$ ln /home/pat/snails /home/mary/snails
```

As is evident from the following `ls` calls, `ln` has entered the file names `frogs` and `snails` in the directory `/home/mary`. The number of links to the files has been incremented by 2 (`ls -l`). The `ls -i` calls show that the file names `frogs` and `snails` have the same inode numbers in both directories.

```
$ ls -l /home/mary
total 6
-rw-rw----   2 pat      proj          34   Mar 09 15:08 frogs
-rw-rw----   2 pat      proj        1217   Mar 09 18:59 snails
$ ls -i /home/pat/frogs /home/mary/frogs
16435 /home/pat/frogs
16435 /home/mary/frogs
$ ls -i /home/pat/snails /home/mary/snails
4766 /home/pat/snails
4766 /home/mary/snails
```

The `ls -l` call above shows that Pat is still the owner of the files. Pat can at any time cancel Mary's right to use the files, e.g. by saying:

```
$ chmod 600 frogs snails
```

If Mary wants to create links with different names, she has to call `ln` twice in **Format 1**:

```
$ ln /home/pat/frogs /home/mary/sugar
$ ln /home/pat/snails /home/mary/spice
```

Example 2

User Norbert needs a quick and simple way to access the directory of his colleague Andrea (`/home2/andrea`) from his own directory (`/home/norbert`). He therefore creates the following symbolic link by calling `ln` in **Format 3**:

```
$ ln -s /home2/andrea /home/norbert/andr
```

Norbert can now use the symbolic link `andr` to directly access Andrea's directory from his own directory, even though her directory resides on another file system:

```
$ ls -lL andr
total 16
drwxr-xr-x   2 andrea   usrother    2560 Feb 27 13:20 PASCAL
drwxr-xr-x   2 andrea   usrother    2048 Mar  5 17:32 COURSES
drwxr-xr-x   2 andrea   usrother     512 Mar  5 12:07 LETTERS
drwxr-xr-x   2 andrea   usrother     512 Feb 26 10:05 bin
-rw-r-xr-x   1 andrea   usrother     148 Feb 20 09:28 testprog
```

Example 3

User Norbert has created the directories `letters89`, `letters90` and `letters91` in his home directory `/home/norbert`:

```
$ ls /home/norbert
letters89  letters90  letters91
```

He now wants direct access from `/home/norbert/letters91` to the other two directories. To do this, he creates two symbolic links by calling `ln` in **Format 4**:

```
$ cd /home/norbert
$ ln -s /home/norbert/letters89 /home/norbert/letters90 letters91
$ cd letters91
$ ls -og letters??
lrwxrwxrwx   1     25 Mar  6 12:42 letters89 -> /home/norbert/letters89
lrwxrwxrwx   1     25 Mar  6 12:42 letters90 -> /home/norbert/letters90
$ ls letters90
offers      letter.henry  letter.nigel  text
```

SEE ALSO

chmod(1), cp(1), find(1), ls(1), mv(1), pax(1), rm(1), link(2), readlink(2), stat(2), symlink(2).

User's Reference Manual

ln(1-ucb)
BSD Compatibility Package

NAME

`ln -` make hard or symbolic links to files

SYNOPSIS

`/usr/ucb/ln` [`–fs`] *filename* [*linkname*]

`/usr/ucb/ln` [`–fs`] *pathname ... directory*

DESCRIPTION

`/usr/ucb/ln` creates an additional directory entry, called a link, to a file or directory. Any number of links can be assigned to a file. The number of links does not affect other file attributes such as size, protections, data, etc.

filename is the name of the original file or directory. *linkname* is the new name to associate with the file or filename. If *linkname* is omitted, the last component of *filename* is used as the name of the link.

If the last argument is the name of a directory, symbolic links are made in that directory for each *pathname* argument; `/usr/ucb/ln` uses the last component of each *pathname* as the name of each link in the named *directory*.

A hard link (the default) is a standard directory entry just like the one made when the file was created. Hard links can only be made to existing files. Hard links cannot be made across file systems (disk partitions, mounted file systems). To remove a file, all hard links to it must be removed, including the name by which it was first created; removing the last hard link releases the inode associated with the file.

A symbolic link, made with the `–s` option, is a special directory entry that points to another named file. Symbolic links can span file systems and point to directories. In fact, you can create a symbolic link that points to a file that is currently absent from the file system; removing the file that it points to does not affect or alter the symbolic link itself.

A symbolic link to a directory behaves differently than you might expect in certain cases. While an `ls(1)` on such a link displays the files in the pointed-to directory, an `ls –l` displays information about the link itself:

```
example% /usr/ucb/ln –s dir link
example% ls link
file1 file2 file3 file4
example% ls –l link
lrwxrwxrwx  1 user            7 Jan 11 23:27 link -> dir
```

When you `cd(1)` to a directory through a symbolic link, you wind up in the pointed-to location within the file system. This means that the parent of the new working directory is not the parent of the symbolic link, but rather, the parent of the pointed-to directory.

For instance, in the following case the final working directory is `/usr` and not `/home/user/linktest`.

```
example% pwd
/home/user/linktest
example% /usr/ucb/ln –s /var/tmp symlink
example% cd symlink
example% cd ..
example% pwd
/usr
```

C shell user's can avoid any resulting navigation problems by using the `pushd` and `popd` built-in commands instead of `cd`.

OPTIONS

`–f` Force a hard link to a directory – this option is only available to the superuser.

–s Create a symbolic link or links.

EXAMPLE

The commands below illustrate the effects of the different forms of the `/usr/ucb/ln` command:

```
example% /usr/ucb/ln file link
example% ls –F file link
file   link
example% /usr/ucb/ln –s file symlink
example% ls –F file symlink
file   symlink@
example% ls –li file link symlink
 10606 -rw-r--r--  2 user            0 Jan 12 00:06 file
 10606 -rw-r--r--  2 user            0 Jan 12 00:06 link
 10607 lrwxrwxrwx  1 user            4 Jan 12 00:06 symlink -> file
example% /usr/ucb/ln –s nonesuch devoid
example% ls –F devoid
devoid@
example% cat devoid
devoid: No such file or directory
example% /usr/ucb/ln –s /proto/bin/* /tmp/bin
example% ls –F /proto/bin /tmp/bin
/proto/bin:
x*      y*      z*
/tmp/bin:
x@      y@      z@
```

NOTES

When the last argument is a directory, simple basenames should not be used for *pathname* arguments. If a basename is used, the resulting symbolic link points to itself:

```
example% /usr/ucb/ln –s file /tmp
example% ls –l /tmp/file
lrwxrwxrwx  1 user            4 Jan 12 00:16 /tmp/file -> file
example% cat /tmp/file
/tmp/file: Too many levels of symbolic links
```

To avoid this problem, use full pathnames, or prepend a reference to the `PWD` variable to files in the working directory:

```
example% rm /tmp/file
example% /usr/ucb/ln –s $PWD/file /tmp
lrwxrwxrwx  1 user
 4 Jan 12 00:16 /tmp/file -> /home/user/subdir/file
```

SEE ALSO

`cp`(1), `ls`(1), `mv`(1), `rm`(1), `link`(2), `readlink`(2), `stat`(2), `symlink`(2).

User's Reference Manual

locale(1)

NAME

`locale -` call up information about the locale

SYNOPSIS

`locale [ –a|–m ] [ –v ]`

`locale [ –l` *locale* `]`

`locale [ –ckv ] [ –l` *locale* `]` *name ...*

DESCRIPTION

The `locale` command writes information about the current locale or other public locales to the standard output. In the following section, a public environment is an environment provided by the implementation that the application can access.

If `locale` is called without any arguments, it summarizes the current locale for each environment category as determined by the settings of the environment variables (see also the XBD specification, Chapter 5, Locale).

If the command is called with operands, it outputs values assigned to the keywords in the environment categories as follows:

- If a keyword name is specified, the keyword as well as the category containing the keyword is output.
- If a category name is specified, the category as well as all the keywords contained in it are output.

OPTIONS

`–a` Outputs information about all available public locales. The locales available include POSIX.

`–c` Outputs the selected environment categories; see **STDOUT**. The `–c` option improves legibility if several categories are selected (e.g. using several keyword names or a category name). This option is valid with or without the specification of the `–k` option.

`–k` Outputs the names and values of the selected keywords.

`–l` *locale*
Outputs information on the public locale *locale*.

`–m` Outputs the character map name.

`–v` Outputs additional titles for the `–a` and `–m` options.

name The name of the environment category as defined in the XBD specification, Chapter 5, Locale, the name of a keyword in an environment category or the reserved name `charmap`. The specified category or the specified keyword can be specified as *name* operands in any order.

ENVIRONMENT VARIABLES

The following environment variables have an effect on the execution of `locale`:

`LANG` Specifies a default value for the locale variable that is unset or null. If `LANG` is unset or null, the corresponding locale default value is used. If the locale variable contains an invalid setting, `locale` behaves as if no variables had been set.

`LC_ALL`
If this value is set, i.e. is not empty, this value overwrites the values of all other locale variables.

`LC_CTYPE`
Determines the locale for the interpretation of byte sequences as characters (e.g. single-byte as opposed to multibyte characters in arguments).

`LC_MESSAGES`
Determines the format and content of error messages.

`NLSPATH`

Determines the position of the message catalog for the processing of LC_MESSAGES.

The LANG, LC_* and NLSPATH environment variables determine the current locale.

STDOUT

If locale is called without options or operands, the names and values of the LANG and LC_* environment variables are written to the standard output. One line is used for every variable, LANG is output first. Only variables set in the environment and not overwritten by LC_ALL are output in the following format:

"%s=%s\n", *variable_name*, *value*

The names of the LC_* variable not set in the environment or overwritten by the LC_ALL variable are output in the following format:

"%s=\"%s\"\n", *variable_name*, *implied_value*

implied_value is the name of the locale selected by the implementation for this category based on the values in LANG and LC_ALL.

value and *implied_value* are enclosed in single quotes so they can be reused as an entry some time in the future. *value* is not enclosed in double quotes (to differentiate it from *implied_value*, which is always enclosed in double quotes).

The LC_ALL variable is output last, using the first format outlined above. If this variable is not set, it is output in the following format:

"LC_ALL=\n"

The following points apply when specifying arguments:

1. If the −a option is set, the names of all the public locales are output in the following format:

   "%s\n", *locale_name*

2. If the −c option is specified, the names of all the selected categories are output in the following format:

   "%s\n", *category_name*

   If keywords were also selected, these are output immediately after the category to which they belong.

   If the −c option is not specified, only the keywords and not the category names are output.

3. If the −k option is specified, the names and values of the selected keywords are output. If the value is non-numeric, it is output in the following format:

   "%s=\"%s\"\n", *keyword_name*, *keyword_value*

   For the charmap keyword, the name of the character map is output if a character map was specified using the localedef −f option during the creation of the locale. charmap is used as *keyword_name* here.

   Numeric values are output in one of the following formats:

   "%s=%d\n", *keyword_name*, *keyword_value*"%s=%c%o\n", *keyword_name*, *escape_character*, *keyword_value*"%s=%cx%x\n", *keyword_name*, *escape_character*, *keyword_value*

   escape_character is the character defined by the escape_char keyword in the current locale (see XBD specification, Section 5.3, Locale Definition).

   Keyword values that are combined (list entries) are separated by semicolons in the output. If the keyword values contain semicolons, double quotes, backslashes, and/or control characters, these characters are escaped using a preceding escape character.

4. If the −k option is not specified, the values of all the keywords are output in the following format:

   "%s\n", *keyword_value*

   For the charmap keyword, the name of the character map is output (if available).

5. If the −m option is specified, a list of all the available character maps is output in the following format:

   "%s\n", *charmap*

The output can be used as an argument for the `localedef –f` option.

EXIT STATUS

`0` All the information requested was found and output.

`>0` An error occurred.

EXAMPLES

The examples are based on the following locales:

```
LANG=locale_x
LC_COLLATE=locale_y
```

The `locale` command would have the following output:

```
LANG='locale_x'
LC_COLLATE='locale_y'
LC_CTYPE="locale_x"
LC_MESSAGES="locale_x"
LC_MONETARY="locale_x"
LC_NUMERIC="locale_x"
LC_TIME="locale_x"
LC_ALL=
```

The `LC_ALL=POSIX locale –ck decimal_point` command would create the following output:

```
LC_NUMERIC
decimal_point="."
```

The following command shows a `locale` application that can be used to determine whether a response entered by a user is a yes-response (affirmation):

```
if printf "%s\n" "$response" | grep -Eq "$(locale yesexpr)"
then
        affirmative processing goes here
else
        non-affirmative processing goes here
fi
```

SEE ALSO

`localedef(1)`.

XBD specification, Section 5.3, Locale Definition

User's Reference Manual

localedef(1)

NAME

`localedef -` define locale

SYNOPSIS

`localedef` [`–bcqs`] [`–d`[*debug-level*]] [`–f` *charmap*] [`–i` *sourcefile*] *name*

DESCRIPTION

The `localedef` command converts source definitions for locales into a format that can be used by the functions and commands whose operational behavior is determined by the setting of the locale variables. This format is defined in the XBD specification, Chapter 5, Locale.

Every Reliant UNIX system user has the facility to create new locales locally. However, only a user with the appropriate privileges (usually system administrators) can install these locales on the system so that they can be used by functions, applications, and commands. This is done either by copying the files created locally to `/usr/lib/locale/`*name* and `/usr/lib/charmap/`*charmap*, or more simply by using the `–s` option.

`localedef` reads source definitions for one or more of the categories in the locales, which belong to the same locale, from the file specified with the `–i` option (if specified), or from the standard input.

The *name* operand indicates the target environment. The command supports the creation of locales which are "public" (i.e. can be accessed generally), as well as locales which are private (i.e. with restricted access rights). On Reliant UNIX systems, only users with the appropriate privileges can create or alter general access locales.

Every category source definition is indicated by the name of the accompanying environment variable and is terminated with an `END`*category-name* instruction. The following categories are supported:

`LC_CTYPE`
: Defines character classification and case conversion.

`LC_COLLATE`
: Defines collation rules.

`LC_MONETARY`
: Defines the format and symbols used for the formatting of monetary information.

`LC_NUMERIC`
: Defines the decimal point, grouping, and "grouping" symbol for the editing of non-monetary numeric information.

`LC_TIME`
: Defines the format and contents of date and time specifications.

`LC_MESSAGES`
: Defines the format and values of yes/no responses (affirmations and negations).

OPTIONS

`–b`
: Suppresses warnings for downwards compatibility. The output created by the `localedef` command can be used by every internationalized Reliant UNIX application, and can also be used by older applications available before the introduction of `localedef`. The internationalization mechanisms of earlier applications were less differentiated than current methods. If a `localedef` source file contains new mechanisms and the `–b` option is not used, warnings are output as the older applications cannot use this new functionality.

`–c`
: Creates an output file, even if warning message is output.

`–d`[*debug-level*]
: Test help messages (grammatical information) are written to the standard output. This option is particularly useful for those developing new locales. The *debug-level* integer value (if present), must not

be separated from the letter "d" by a blank.

`-f` *charmap*
: Specifies the pathname of a file which contains an assignment of the symbolic character symbols and collating element symbols to the actual character coding (character map). The *charmap* format is described in the XBD specification, Section 4.4, Character Set Description File. This option must be specified if symbolic names are used that were not defined using the `collating-symbol` key word. If the `-f` option is not specified, the character assignment defined in the `/usr/lib/charmap/posix` file (ISO 8859-1) is used.

`-i` *inputfile*
: The pathname of a file which contains the source definitions. If this option is not specified, the source definitions of the default input are read. The format of the input file is described in the XBD specification, Section 5.3, Locale Definition.

`-q`
: No output is written to the standard output.

`-s`
: Option for the system administrator. Installs the generated locale (and possibly the new *charmap* file) in `/usr/lib/locale/`*name* (or in `/usr/lib/charmap/`*charmap*).

name
: Specifies the locale. The use of this name is described in the XBD specification, Chapter 5, Locale.
  - If *name* contains one or more slashes, *name* is interpreted as a pathname, in which the definitions created for the locales are saved.
  - If *name* does not contain a slash and the `-s` option is used, the locale can be accessed generally and is generated under `/usr/lib/locale/`*name*. This facility is restricted to users with the appropriate privileges.
  - If *name* does not contain a slash and the `-s` option is not used, the locale is private, and is generated in the current directory.

  Because only one name can be specified, the only categories that can be processed in a call are those that belong to the same locale.

STANDARD INPUT

If the `-i` option is not specified, the standard input must be a text file containing one or more source definitions for categories for locales, as described in the XBD specification, Section 5.3, Locale Definition. If lines are continued using the escape mechanism, there is no length restriction for the entire input line.

INPUT FILES

The file containing the description of a character set (*charmap* argument in `-f`) is described in the XBD specification, Section 4.4, Character Set Description File. If a source definition of a category for an locale contains the `copy` instruction (as described in the XBD specification, Chapter 5, Locale) and this instruction specifies a valid locale, `localedef` behaves as if the source definition contained a valid source definition of the category for the locale specified.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of `localedef`:

`LANG`
: Specifies a default value for the locale variables that are unset or null. If `LANG` is unset or null, the corresponding default value from the locale is used. If the locale variable contains an invalid setting, the command behaves as if none of the variables had been defined.

`LC_ALL`
: If this variable is set, i.e. it is not empty, this value overwrites the values of all other variables for the locale.

`LC_COLLATE`
: (This variable does not affect `localedef`; the POSIX locale is used for this category.)

`LC_CTYPE`
: Determines the locale for the interpretation of sequences of bytes (e.g. single-byte as opposed to

multibyte characters in arguments and input files). This variable has no effect on the processing of command input files. The POSIX locale is used here, regardless of the value of the variable.

`LC_MESSAGES`
Determines the format and contents of messages.

`NLSPATH`
Determines the location of message catalogs for the processing of `LC_MESSAGES`.

OUTPUT FILES

The format of the created output corresponds to the format of the internationalization database on Reliant UNIX systems:

name (if slashes are contained) or `/usr/lib/locale/`*name* (without slashes and with the `-s` option) or `./`*name* (without slashes and without the `-s` option) is the pathname of a directory containing the following newly generated files:

`LC_COLLATE`
File with collating information

`LC_CTYPE`
File with information on the character type (`ctype`)

`LC_MONETARY`
File with monetary information

`LC_NUMERIC`
File with numeric information

`LC_TIME`
File with date and time information

`LC_MESSAGES`
Directory with message information

`LC_MESSAGES/Xopen_info`
File with yes/no information

`loc_info`
File with information on the name and path of the character map

A copy of the character map is also created in `/usr/lib/charmap` if the `-f` and `-s` options are used.

EXIT STATUS

The following exit values are transferred:

`0` No errors occurred and the locales were successfully created.

`1` Warnings were output, but the locales were created successfully nonetheless.

`2` The locale specification exceeded implementation limits or the coded character set or sets used were not supported by the implementation. No locale was created.

`4` Warnings occurred and no output was created.

CONSEQUENCES OF ERRORS

If an error is detected in the input file, no output file is created.

If warnings occur, an output file is created provided that the `-c` option was specified. The following conditions will cause warning messages to be output:

- A symbolic name not contained in the *charmap* file was used for the description of the `LC_CTYPE` or `LC_COLLATE` categories (for other categories this causes an error condition).
- The number of operands for the `order` keyword exceeds the {`COLL_WEIGHTS_MAX`} limit.
- The source contains optional keywords which are not supported by the implementation.

APPLICATION USAGE

The character map definition is optional, and is not contained in the locale definition. This allows both completely self-defined source files and generic sources (applicable to more than one coded character set). To aid portability all character map definitions must have the same symbolic name for portable character sets. As described in the XBD specification, Section 4.4, Character Set Description File, it depends on the relevant implementation whether users or applications can use additional character set description files. Therefore, the `-f` option might only operate correctly if a *charmap* supported by the implementation is specified.

SEE ALSO

`locale`(1).

X/Open CAE Specification. *System Interface Definitions*, *Issue 4* (XBD). ISBN: 1-872630-46-4

User's Reference Manual

logger(1)

NAME

`logger` - log messages

SYNOPSIS

`logger` *string*

DESCRIPTION

The `logger` utility saves a message, in an unspecified manner and format, containing the string operands provided by the user. The messages are expected to be evaluated later by personnel performing system administration tasks.

It is implementation-dependent whether messages written in locales other than the POSIX locale are effective.

OPTIONS

string
: One of the string arguments whose contents are concatenated together, in the order specified, separated by single space characters.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of `logger`:

`LANG` Provide a default value for the internationalization variables that are unset or null. If `LANG` is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalization variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

`LC_ALL`
: If set to a non-empty string value, override the values of all the other internationalization variables.

`LC_CTYPE`
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

`LC_MESSAGES`
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error. (This means diagnostics from `logger` to the user or application, not diagnostic messages that the user is sending to the system administrator).

EXIT STATUS

The following exit values are returned:

`0` Successful completion.

`>0` An error occurred.

APPLICATION USAGE

This utility allows logging of information for later use by a system administrator or programmer in determining why non-interactive utilities have failed. The locations of the saved messages, their format and retention period are all unspecified. There is no method for a portable application to read messages, once written.

EXAMPLE

A batch application, running non-interactively, tries to read a configuration file and fails; it may attempt to notify the system administrator with:

```
logger myname: unable to read file foo. [timestamp]
```

SEE ALSO

mailx(1), write(1).

User's Reference Manual

logger(1-ucb)
BSD Compatibility Package

NAME

`logger -` add entries to the system log

SYNOPSIS

`/usr/ucb/logger` [`–t` *tag*] [`–p` *priority*] [`–i`] [`–f` *filename*] [*message*] ...

DESCRIPTION

`logger` provides a method for adding one-line entries to the system log file from the command line. One or more *message* arguments can be given on the command line, in which case each is logged immediately. Otherwise, a *filename* can be specified, in which case each line in the file is logged. If neither is specified, `logger` reads and logs messages on a line-by-line basis from the standard input.

OPTIONS

`–t` *tag*
: Mark each line added to the log with the specified *tag*.

`–p` *priority*
: Enter the message with the specified *priority*. The message priority can be specified numerically, or as a *facility*`.`*level* pair. For example, `–p local3.info` assigns the message priority to the `info` level in the `local3` facility. The default priority is `user.notice`.

`–i`
: Log the process ID of the `logger` process with each line.

`–f` *filename*
: Use the contents of *filename* as the message to log.

message
: If this is unspecified, either the file indicated with `–f` or the standard input is added to the log.

EXAMPLE

```
logger System rebooted
```

will log the message "`System rebooted`" to the facility at priority `notice` to be treated by `syslogd` as other messages to the facility `notice` are.

```
logger –p local0.notice –t HOSTIDM –f /dev/idmc
```

will read from the file `/dev/idmc` and will log each line in that file as a message with the tag "`HOSTIDM`" at priority `notice` to be treated by `syslogd` as other messages to the facility `local0` are.

SEE ALSO

`syslogd`(1M), `syslog`(3C).

login(1)

NAME

`login` - system login

SYNOPSIS

`login` [*login_name* [*environment ...*]]

DESCRIPTION

In the Korn shell the `login` command is a shell built-in. In the Bourne shell `/usr/bin/login` is invoked.

`login` can be called by the user and by the system:

- When you first sign in on a terminal, `login` is invoked by the system.
- If you wish to call `login` as a command in the Bourne shell, it must replace the current login shell. This entails invoking the command from the login shell in the form `exec login`, which terminates the current shell when `login` is started. The fact that the process ID does not change indicates that no new process is created.

`login` performs the following actions:

- it prompts for the password associated with the specified login name.
- it makes the home directory the working directory. The login directory that is entered in the `/etc/passwd` file for the given login name is assigned to the `HOME` variable.
- it initializes the standard shell variables to the appropriate default values (see **Functionality of the** `login` command).
- it invokes the startup program that is assigned to the given login name in the `/etc/passwd` file.

If the startup program assigned to the specified login name in the `/etc/passwd` file is `/usr/bin/sh`, `login` is overlaid by a new login shell which performs the following actions:

- it executes the file `/etc/profile`.
- it also executes the file `$HOME/.profile` if you have created one.

The shell environment defined for the given login name will thus be valid from this point on. If you wish to retain your current shell environment and directory when switching to another login name, you should use the `su(1)` command instead of `login`.

If the line `CONSOLE=`*terminal* is defined in the `/etc/default/login` file, the only terminal on which you can log in as `root` is the terminal defined as the console.

Before the call

The name you use to log in must appear in the `/etc/passwd` file. If it does not, you should contact the system administrator.

If there are no lowercase letters in the first line of input processed, `login` assumes that the terminal can only process uppercase letters.

login_name

Login name under which you wish to log in. This name must appear in the `/etc/passwd` file.

login_name not specified:

The `login` command will prompt you to enter the login name in the next line.

If a password has been defined for this login name in the `/etc/shadow` file, `login` will then prompt you to enter the password.

Your input will not be displayed on the screen or recorded in a file.

environment

Arguments that can be specified in the form *yyy* or *xxx*=*yyy* to assign a value to an environment variable.

These arguments may be supplied either at execution time or when you enter your login name. *xxx* is the name of an environment variable; *yyy* is a value to be assigned to an environment variable. Typing a backslash in front of a character in *yyy* quotes that character, thus allowing characters such as spaces and tabs to be included.

xxx=yyy
: Environment variable *xxx* is assigned the value *yyy*. If the environment variable already has a value, it is replaced by the new value, with two exceptions: the variables `PATH` and `SHELL` cannot be changed in this way.

yyy
: The value *yyy* is assigned to the environment variable `L`*n*, where *n* is a number that starts at 0 and is incremented each time a variable name is required (until all *yyy* values have been assigned).

Functionality of the login command

The `login` command compares the specified login name and password with the corresponding entry in the `/etc/passwd` and `/etc/shadow` files. If all entries are accurate, you will be logged into the system under the given login name.

If invalid entries are made, the terminal line will be disconnected after five unsuccessful login attempts. This will also occur if you do not complete the login successfully within a defined period of time.

The following steps are performed sequentially after a successful login:

- `login` searches the `/etc/passwd` file for the user ID (UID) and group ID (GID) associated with the specified login name.

  Login names are internally maintained by the operating system under their user and group IDs only.
- `login` enters the login name in the `/var/adm/utmp` file. This file contains information about all users currently logged in.

  The `/var/adm/utmp` file is accessed by the `who(1)` command.
- If the `/var/adm/wtmp` file exists, `login` enters the login name in it. This file is used to consecutively record all attempts to log in and out.
- `login` displays the last time you logged in under this login name. This information is obtained from the file `/var/adm/lastlog`.
- `login` assigns default values to the following environment variables:

Variable	Assignment:
`HOME`	Name of the login directory (from `/etc/passwd`)
`LOGNAME`	Login name (from `/etc/passwd`)
`MAIL`	`/var/spool/mail/`*login_name*
`SHELL`	`/usr/bin/sh` (from `/etc/passwd`)
`TZ`	defined timezone (from `/etc/default/login` or `/etc/TIMEZONE`)
`PATH`	defined search path (from `/etc/default/login`)
`LANG`	NLS variable for defining an internationalized environment
`TTY`	Terminal port number
`TERM`	Terminal type
`USER`	User's login name

You will find more information on these environment variables in the description of the `sh` command. Here you will also find out how to change default values and define additional environment variables.

- `login` switches to the login directory that is entered in `/etc/passwd` file for this login name.

- `login` invokes the program that is assigned to the given login name in the `/etc/passwd` file. Typically, `/usr/bin/sh` is started as the login shell [see sh(1)]. This shell overlays `login`.

  If the program name `/etc/passwd` is given as *, the named login directory becomes the root directory, thus making it the starting point for path searches for file names that begin with a / (slash). In addition, `login` is re-executed at the new root structure.

Starting work at the terminal

When you initially turn on the terminal, the cursor appears on the screen. The system then prompts you for a login name (on some terminals you may have to press CTRL-D first) and passes your input to `login`. `login` then prompts you for a password. (If you log in incorrectly, `login` again prompts you for a login name.) This also applies after you have terminated a session by pressing CTRL-D.

The following system processes must thus be active before you can begin working at the terminal:

- `/etc/init` starts the system process `getty` for all connected terminals. A new process is thus generated for each terminal.
- `/etc/getty` displays the welcome screen and overlays itself with `login` after reading the login name.

After you have successfully logged in, `login` overlays itself with the startup program entered in the `/etc/passwd` file. This is normally the login shell `/usr/bin/sh`. If you terminate this login shell with CTRL-D, you will terminate the current process, i.e. `/etc/init` restarts the system process `/etc/getty`.

ERROR MESSAGES

`Login incorrect`

This error message can have the following causes:

- You have made a typing error when entering the login name or the password. The `login` command prompts you to repeat your input.
- This login name has not yet been set up. Contact the system administrator.
- The password has been changed. Contact the system administrator.

`No shell`

You have either no execute permission for the corresponding shell under this login name, or the entry in the `/etc/passwd` file has not been correctly made for this login name. Contact the system administrator.

`unable to change directory to ""`

The login directory for this login name has not been installed. Contact the system administrator.

`Cannot open password file`

The `/etc/passwd` file does not exist or cannot be read by `login`. Contact the system administrator.

`No utmp entry. You must execute "exec login" from the lowest level "sh".`

`login` cannot find the right entry in `/var/adm/utmp`, which means that `login` has not been started as a login process. Call `exec login` from the login shell.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. The `LC_TIME` environment variable governs the format of date and time strings. `LC_CTYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES`, `LC_TIME` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables which affect internationalization. If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

It is advisable to restrict the characters you use for login names and passwords to those in the Portable

Filename Character Set; 8-bit data may not always be portable to non-internationalized systems.

EXAMPLES

Example 1

User `rose` is currently working in the Bourne shell and wishes to log into the system again under the login name `harold`. The shell prompt has been redefined in the file `/home/harold/.profile` as follows:

```
PS1=$USER
```

```
$ pwd
/usr/rose/prog
$ id
uid=104(rose) gid=12(group12)
$ exec login harold
password:
```

The password for `harold` must now be entered unseen. Following a successful login attempt, `login` displays information about the time and terminal of the last login.

```
harold: pwd
/home/harold
```

Example 2

The `su` command can be used to switch the login name without affecting the current working environment:

```
$ pwd
/usr/rose/prog
$ id
uid=104(rose) gid=12(group12)
$ su harold
password:
```

The password for harold must now be entered unseen.

```
$ pwd
/usr/rose/prog
$ echo $USER
rose
$ id
id=100(harold) gid=10(other)
```

NOTES

`login` exists both as an external command (`/usr/bin/login`) and as a built-in shell command in the Korn shell `ksh(1)` . The shell generates a new process to execute `/usr/bin/login`.

Some differences in behavior may occur when using `login`, depending on which command is being used. The possible differences are not described specifically.

FILES

`/etc/motd`
: Contains the login message (message of the day) for all users.

`/etc/nologin`
: If this file exists, only the system administrator (`root`) can log in. If another user attempts to log in, the contents of the file `/etc/nologin` are output.

`/etc/passwd`
: Contains all the login names that have been set up.

`/etc/profile`
: Executed by each login shell. This file is created by the system administrator.

`/etc/shadow`
: Contains encoded passwords.

`/var/adm/utmp`
: Contains information on logins by all users currently logged in.

`/var/adm/loginlog`
: Contains information on unsuccessful login attempts.

`/var/adm/lastlog`
: Records for each login name the time and terminal at which the respective user last logged in.

`/var/adm/wtmp`
: If this file exists, all attempts to log in or out are sequentially entered in it.

`/var/mail/`*login_name*
: Contains mail for this login name.

`$HOME/.profile`
: Shell script that every user can set up in his or her home directory. This file is executed by the login shell after `/etc/profile`.

  Note:

  The C shell does not execute `$HOME/.profile`; see `csh(1)`.

SEE ALSO

`exec(1)`, `newgrp(1)`, `sh(1)`, `su(1)`, `default_passwd(4)`, `loginlog(4)`, `passwd(4)`, `profile(4)`, `environ(5)`.

User's Reference Manual

logname(1)

NAME

`logname` - display login name

SYNOPSIS

```
logname
```

DESCRIPTION

`logname` writes the user's login name to standard output. The login name is the string that will be returned by the `getlogin(3C)` function. If `getlogin(3C)` fails, a further attempt is made with the `cuserid(3S)` function and the following message is displayed on standard error: `getlogin() call failed, trying cuserid()`.

Interaction with su

If you have logged in under a login name *name*, and then change to another login name with `su` and enter `logname` under this new identity, `logname` outputs the login name `name` in the same way as the `who` or `who am i` command. `id`, on the other hand, outputs the current login name according to `su`.

EXAMPLES

Example 1

You log in under the login name `peter` and then display this login name:

```
$ logname
peter
```

Example 2

You now switch to the login name `david` with `su` and then display the login name with `logname`, `who am i` and `id`:

```
$ su david
$ logname
peter
$ who am i
peter       tty004       Mar  7  18:59
$ id
uid=1097(david) gid=20(user)
```

NOTES

`utmp` logging is made available with some X11 emulations. If the user waives this option, `logname` cannot return a correct value (error message: "`getlogin() call failed,trying cuserid()`").

FILES

`/etc/profile`
: File that is evaluated by each login shell. It is used for setting a shell environment, depending on the contents of the `LOGNAME` (login name of the user) environment variable passed by `login`. This file is created by the system administrator.

SEE ALSO

`env(1)`, `id(1)`, `login(1)`, `su(1)`, `who(1)`, `getlogin(3C)`, `cuserid(3S)`, `environ(5)`.

User's Reference Manual

look(1)
BSD Compatibility Package

NAME

`look` - find lines in a sorted list

SYNOPSIS

`/usr/ucb/look` [`–df`] *string* [*file*]

DESCRIPTION

`look` consults a sorted *file* and prints all lines that begin with *string*. It uses binary search.

OPTIONS

The options `–d` and `–f` affect comparisons as in `sort(1)`:

`–d` "Dictionary" order: only letters, digits, tabs, and blanks participate in comparisons.

`–f` Fold. Uppercase letters compare equal to lowercase.

If no *file* is specified, `/usr/ucblib/dict/words` is assumed with collating sequence `–df`.

FILES

`/usr/ucblib/dict/words`

SEE ALSO

`grep(1)`, `sort(1)`.

User's Reference Manual

lookbib(1)
BSD Compatibility Package

NAME

`lookbib` - find references in a bibliographic database

SYNOPSIS

`/usr/ucb/lookbib` *database*

DESCRIPTION

A bibliographic reference is a set of lines, constituting fields of bibliographic information. Each field starts on a line beginning with a "`%`", followed by a key-letter, then a blank, and finally the contents of the field, which may continue until the next line starting with "`%`" [see `addbib(1)`].

`lookbib` uses an inverted index made by `indxbib` to find sets of bibliographic references. It reads keywords typed after the "`>`" prompt on the terminal, and retrieves records containing all these keywords. If nothing matches, nothing is returned except another "`>`" prompt.

It is possible to search multiple databases, as long as they have a common index made by `indxbib`. In that case, only the first argument given to `indxbib` is specified to `lookbib`.

If `lookbib` does not find the index files (the `.i[abc]` files), it looks for a reference file with the same name as the argument, without the suffixes. It creates a file with a `.ig` suffix, suitable for use with `fgrep(1)`. `lookbib` then uses this `fgrep` file to find references. This method is simpler to use, but the `.ig` file is slower to use than the `.i[abc]` files, and does not allow the use of multiple reference files.

NOTE

Probably all dates should be indexed, since many disciplines refer to literature written in the 1800s or earlier.

FILES

`*.ia` entry file
`*.ib` posting file
`*.ic` tag file
`*.ig` reference file

SEE ALSO

`addbib(1)`, `grep(1)`, `indxbib(1)`, `refer(1)`, `roffbib(1)`, `sortbib(1)`.

User's Reference Manual

lp(1)
AT&T Spooler only

NAME

`lp, cancel -` send/cancel requests to an LP printservice

SYNOPSIS

`lp` [*printing-options*] [*files*]

`lp -i` *request-IDs printing-options*

`cancel` [*request-IDs*] [*printers*]

`cancel –u` *login-ID-list* [*printers*]

DESCRIPTION

The first form of the `lp` command arranges for the named *files* and associated information (collectively called a *request*) to be printed. If no file names are specified on the command line, the standard input is assumed. The standard input may be specified along with named *files* on the command line by listing the file name(s) and specifying – for the standard input. The *files* will be printed in the order in which they appear on the shell command line.

The LP print service associates a unique *request-ID* with each request and displays it on the standard output. This *request-ID* can be used later when canceling or changing a request, or when determining its status. [See the section on `cancel` for details about canceling a request, and `lpstat(1)` for information about checking the status of a print request.]

The second form of `lp` is used to change the options for a request. The print request identified by the *request-ID* is changed according to the printing options specified with this shell command. The printing options available are the same as those with the first form of the `lp` shell command. If the request has finished printing, the change is rejected. If the request is already printing, it will be stopped and restarted from the beginning (unless the `–P` option has been given).

The `cancel` command allows users to cancel print requests previously sent with the `lp` command. The first form of `cancel` permits cancellation of requests based on their *request-ID*. The second form of cancel permits cancellation of requests based on the *login-ID* of their owner.

Sending a Print Request

The first form of the `lp` command is used to send a print request to a particular printer or group of printers.

Options to `lp` must always precede file names, but may be specified in any order. The following options are available for `lp`:

`–c` Make copies of the *files* before printing. Normally, *files* will not be copied, but will be linked whenever possible. If the `–c` option is not given, then the user should be careful not to remove any of the *files* before the request has been printed in its entirety. It should also be noted that if the `–c` option is not specified, any changes made to the named *files* after the request is made but before it is printed will be reflected in the printed output.

`–d` *dest*
Choose *dest* as the printer or class of printers that is to do the printing. If *dest* is a printer, then the request will be printed only on that specific printer. If *dest* is a class of printers, then the request will be printed on the first available printer that is a member of the class. If *dest* is `any`, then the request will be printed on any printer which can handle it. Under certain conditions (unavailability of printers, file space limitations, and so on) requests for specific destinations may not be accepted [see `lpstat(1)`]. If `–d` is not set, the printer or printer type of the `LPDEST`, or alternatively, the `PRINTER` environment variable, is assumed (provided one of them has been set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems [see `lpstat(1)`].

`-f` *form-name* [`-d any`]
Print the request on the form *form-name*. The LP print service ensures that the form is mounted on the printer. If *form-name* is requested with a printer destination that cannot support the form, the request is rejected. If *form-name* has not been defined for the system, or if the user is not allowed to use the form, the request is rejected [see `lpforms(1M)`]. When the `-d any` option is given, the request is printed on any printer that has the requested form mounted and can handle all other needs of the print request.

`-H` *special-handling*
Print the request according to the value of *special-handling*. Acceptable values for *special-handling* are defined below:

`hold` Don't print the request until notified. If printing has already begun, stop it. Other print requests will go ahead of a held request until it is resumed.

`resume`
Resume a held request. If it had been printing when held, it will be the next request printed, unless subsequently bumped by an `immediate` request.

`immediate`
(Available only to LP administrators) Print the request next. If more than one request is assigned `immediate`, the requests are printed in the reverse order queued. If a request is currently printing on the desired printer, you have to put it on hold to allow the immediate request to print.

`-m` Send mail [see `mail(1)`] after the files have been printed. By default, no mail is sent upon normal completion of the print request.

`-n` *number*
Print *number* copies (default is 1) of the output.

`-o` *option*
Specify printer-dependent *options*. Several such *options* may be collected by specifying the `-o` keyletter more than once (`-o` $option_1$ `-o` $option_2$ ... `-o` $option_n$), or by specifying a list of options with one `-o` keyletter enclosed in double quotes (that is, `-o "`$option_1$ $option_2$ ... $option_n$`"`). The standard interface recognizes the following options:

`nobanner`
Do not print a banner page with this request. (The administrator can disallow this option at any time.)

`nofilebreak`
Do not insert a form feed between the files given, if submitting a job to print more than one file.

`length=`*scaled-decimal-number*
Print this request with pages *scaled-decimal-number* lines long. A *scaled-decimal-number* is an optionally scaled decimal number that gives a size in lines, columns, inches, or centimeters, as appropriate. The scale is indicated by appending the letter "i" for inches, or the letter "c" for centimeters. For length or width settings, an unscaled number indicates lines or columns; for line pitch or character pitch settings, an unscaled number indicates lines per inch or characters per inch (the same as a number scaled with "i"). For example, `length=66` indicates a page length of 66 lines, `length=11i` indicates a page length of 11 inches, and `length=27.94c` indicates a page length of 27.94 centimeters.

This option may not be used with the `-f` option.

`width=`*scaled-decimal-number*
Print this request with page-width set to *scaled-decimal-number* columns wide. (See the explanation of *scaled-decimal-numbers* in the discussion of `length`, above.) This option may not be used with the `-f` option.

`lpi=`*scaled-decimal-number*
: Print this request with the line pitch set to *scaled-decimal-number* lines per inch. This option may not be used with the `-f` option.

`cpi=`*scaled-decimal-number*
: Print this request with the character pitch set to *scaled-decimal-number* characters per inch. Character pitch can also be set to `pica` (representing 10 characters per inch) or `elite` (representing 12 characters per inch), or it can be `compressed` (representing as many characters as a printer can handle). There is no standard number of characters per inch for all printers; see the Terminfo database [`terminfo(4)`] for the default character pitch for your printer.

  This option may not be used with the `-f` option.

`stty='`*stty-option-list*`'`
: A list of options valid for the `stty` command; enclose the list with single quotes if it contains blanks.

`-P` *page-list*
: Print the pages specified in *page-list*. This option can be used only if there is a filter available to handle it; otherwise, the print request will be rejected.

  The *page-list* may consist of range(s) of numbers, single page numbers, or a combination of both. The pages will be printed in ascending order.

`-q` *priority-level*
: Assign this request *priority-level* in the printing queue. The values of *priority-level* range from 0, the highest priority, to 39, the lowest priority. If a priority is not specified, the default for the print service is used, as assigned by the system administrator. A priority limit may be assigned to individual users by the system administrator.

`-s`
: Suppress messages from `lp` such as those that begin with `request id is`.

`-S` *character-set* [`-d any`]
`-S` *print-wheel* [`-d any`]
: Print this request using the specified *character-set* or *print-wheel*. If a form was requested and it requires a character set or print wheel other than the one specified with the `-S` option, the request is rejected.

  For printers that take print wheels: if the print wheel specified is not one listed by the administrator as acceptable for the printer specified in this request, the request is rejected unless the print wheel is already mounted on the printer.

  For printers that use selectable or programmable character sets: if the *character-set* specified is not one defined in the Terminfo database for the printer [see `terminfo(4)`], or is not an alias defined by the administrator, the request is rejected.

  When the `-d any` option is used, the request is printed on any printer that has the print wheel mounted or any printer that can select the character set, and that can handle any other needs of the request.

`-t` *title*
: Print *title* on the banner page of the output. The default is no title. Enclose *title* in quotes if it contains blanks.

`-T` *content-type* [`-r`]
: Print the request on a printer that can support the specified *content-type*. If no printer accepts this type directly, a filter will be used to convert the content into an acceptable type. If the `-r` option is specified, a filter will not be used. If `-r` is specified, and no printer accepts the *content-type* directly, the request is rejected. If the *content-type* is not acceptable to any printer, either directly or with a filter, the request is rejected.

`-w`
: Write a message on the user's terminal after the *files* have been printed. If the user is not logged in, then mail will be sent instead.

`-y` *mode-list*

Print this request according to the printing modes listed in *mode-list*. The allowed values for *mode-list* are locally defined. This option may be used only if there is a filter available to handle it; otherwise, the print request will be rejected.

`--` Must be placed in front of the filename if it begins with `-`.

Canceling a Print Request

The `cancel` command cancels requests for print jobs made with the `lp` command. The first form allows a user to specify one or more *request-IDs* of print jobs to be canceled. Alternatively, the user can specify one or more *printers*, on which only the currently printing job will be canceled.

The second form of `cancel` permits a user to cancel all of his or her own jobs on all printers. In this form the *printers* option can be used to restrict the printers on which the user's jobs will be canceled. Note that in this form, when the *printers* option is used, all jobs queued for those printers will be canceled. A printer class is not a valid argument.

Users without special privileges can cancel only requests associated with their own login IDs. The system administrator can cancel jobs submitted by any user. The *login-ID-list* must be enclosed in quotes if it contains blanks.

NOTES

Printers for which requests are not being accepted will not be considered when the `lp` command is run and the destination is `any`. (Use the `lpstat -a` command to see which printers are accepting requests.) On the other hand, if (1) a request is destined for a class of printers and (2) the class itself is accepting requests, then *all* printers in the class will be considered, regardless of their acceptance status.

For printers that take mountable print wheels or font cartridges, if you do not specify a particular print wheel or font with the `-S` option, whichever one happens to be mounted at the time your request is printed will be used. Use the `lpstat -p` *printer* `-l` command to see which print wheels are available on a particular printer, or the `lpstat -S -l` command to find out what print wheels are available and on which printers. For printers that have selectable character sets, you will get the standard character set if you don't use the `-S` option.

INTERNATIONALIZED ENVIRONMENT

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. `LC_TIME` governs the format and contents of date specifications in the cover page (if available).

If `LC_MESSAGES` or `LC_TIME` is undefined or defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all other environment variables that affect internationalization.

ENVIRONMENT VARIABLES

`LPDEST`
: Defines the printer or printer type to which the output is to be written. `LPDEST` is only evaluated if the `-d` option is not set.

`PRINTER`
: Defines the printer or printer type to which the output is to be written. `PRINTER` is only evaluated if the `-d` option is not set and `LPDEST` has not been assigned a value.

FILES

`/var/spool/lp/*`

SEE ALSO

`enable(1)`, `lpstat(1)`, `mail(1)`, `accept(1M)`, `lpadmin(1M)`, `lpfilter(1M)`, `lpforms(1M)`, `lpsched(1M)`, `lpsystem(1M)`, `lpusers(1M)`, `terminfo(4)`.

User's Reference Manual

lpq(1)
BSD Compatibility Package

NAME

`lpq` - display the queue of printer jobs

SYNOPSIS

`/usr/ucb/lpq` [`-P`*printer*] [`-l`] [+[*interval*]] [*job#* ...] [*username* ...]

DESCRIPTION

`lpq` displays the contents of a printer queue. It reports the status of jobs specified by *job#*, or all jobs owned by the user specified by *username*. `lpq` reports on all jobs in the default printer queue when invoked with no arguments.

For each print job in the queue, `lpq` reports the user's name, current position, the names of input files comprising the job, the job number [by which it is referred to when using `lprm(1)`] and the total size in bytes. Normally, only as much information as will fit on one line is displayed. Jobs are normally queued on a first-in-first-out basis. Filenames comprising a job may be unavailable, such as when `lpr` is used at the end of a pipeline; in such cases the filename field indicates the standard input.

If `lpq` warns that there is no daemon present (that is, due to some malfunction), the `lpc(1M)` command can be used to restart a printer daemon.

OPTIONS

`-P` *printer*
: Display information about the queue for the specified *printer*. In the absence of the `-P` option, the queue to the printer specified by the `PRINTER` variable in the environment is used. If the `PRINTER` variable is not set, the queue for the default printer is used.

`-l`
: Display queue information in long format; includes the name of the host from which the job originated.

+[*interval*]
: Display the spool queue periodically until it empties. This option clears the terminal screen before reporting on the queue. If an *interval* is supplied, `lpq` sleeps that number of seconds in between reports.

DIAGNOSTICS

`printer is printing`
: The `lpq` program queries the spooler `lpsched` about the status of the printer. If the printer is disabled, the superuser can restart the spooler using `lpc(1M)` .

`printer waiting for auto-retry (offline ?)`
: The daemon could not open the printer device. The printer may be turned off-line. This message can also occur if a printer is out of paper, the paper is jammed, and so on. Another possible cause is that a process, such as an output filter, has exclusive use of the device. The only recourse in this case is to kill the offending process and restart the printer with `lpc`.

`waiting for` *host* `to come up`
: A daemon is trying to connect to the remote machine named *host*, in order to send the files in the local queue. If the remote machine is up, `lpd` on the remote machine is probably dead or hung and should be restarted using `lpc`.

`sending to` *host*
: The files are being transferred to the remote *host*, or else the local daemon has hung while trying to transfer the files.

`printer disabled reason:`
: The printer has been marked as being unavailable with `lpc`.

`lpq: The LP print service isn't running or can't be reached.`
The `lpsched` process overseeing the spooling queue does not exist. This normally occurs only when the daemon has unexpectedly died. You can restart the printer daemon with `lpc`.

`lpr:` *printer*: unknown printer
The `printer` was not found in the System V LP database. Usually this is a typing mistake; however, it may indicate that the printer does not exist on the system. Use `lptstat –p` to find the reason.

`lpr: error on opening queue to spooler`
The connection to `lpsched` on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check if the printer spooler daemon `/usr/lib/lpsched` is running.

`lpr: Can't send message to LP print service`
`lpr: Can't receive message from LP print service`
These indicate that the LP print service has been stopped. Get help from the system administrator.

`lpr: Received unexpected message from LP print service`
It is likely there is an error in this software. Get help from system administrator.

NOTE

Output formatting is sensitive to the line length of the terminal; this can result in widely-spaced columns.

FILES

`/var/spool/lp`
spooling directory

`/var/spool/lp/tmp/`*system_name*`/*-0`
request files specifying jobs

SEE ALSO

`lp(1)`, `lpr(1)`, `lprm(1)`, `lpc(1M)`, `lpsched(1M)`.

lpr(1)
BSD Compatibility Package

NAME

`lpr` - send a job to the printer

SYNOPSIS

`/usr/ucb/lpr` [`–P` *printer*] [`–#` *copies*] [`–C`*class*] [`–J` *job*] [`–T` *title*] [`–i`[*indent*]] [`–w` *cols*] .ift.ti +u [`–B`] [`–r`] [`–m`] [`–h`] [`–s`] [*–filter_option*] [*filename ...*]

DESCRIPTION

`lpr` forwards printer jobs to a spooling area for subsequent printing as facilities become available. Each printer job consists of copies of, or, with `–s`, complete pathnames of each *filename* you specify. The spool area is managed by the line printer spooler, `lpsched`. `lpr` reads from standard input if no files are specified.

OPTIONS

`–P` *printer*
: Send output to the named *printer*. Otherwise send output to the printer named in the `PRINTER` environment variable, or to the default printer, `lp`.

`–#` *copies*
: Produce the number of *copies* indicated for each named file. For example:

```
lpr –#3 index.c lookup.c
```

produces three copies of `index.c`, followed by three copies of `lookup.c`. On the other hand,

```
cat index.c lookup.c | lpr –#3
```

generates three copies of the concatenation of the files.

`–C` *class*
: Print *class* as the job classification on the first page. For example,

```
lpr –C Operations new.index.c
```

replaces the system name (the name returned by *hostname*) with `Operations` on the first page, and prints the file `new.index.c`.

`–J` *job*
: Print *job* as the job name on the burst page. Normally, `lpr` uses the first file's name.

`–T` *title*
: Use *title* instead of the file name for the title used by `pr(1)` .

`–i`[*indent*]
: Indent output *indent* SPACE characters. Eight SPACE characters is the default.

`–w` *cols*
: Use *cols* as the page width for `pr`.

`–r`
: Remove the file upon completion of spooling, or upon completion of printing with the `–s` option. This is not supported in the SunOS compatibility package. However if the job is submitted to a remote SunOS system, these options will be sent to the remote system for processing.

`–m`
: Send mail upon completion.

`–h`
: Suppress printing the burst page.

`–s`
: Use the full pathnames (not symbolic links) of the files to be printed rather than trying to copy them. This means the data files should not be modified or removed until they have been printed. `–s` only prevents copies of local files from being made. Jobs from remote hosts are copied anyway. `–s` only works with named data files; if the `lpr` command is at the end of a pipeline, the data is copied to the spool.

filter_option

The following single letter options notify the line printer spooler that the files are not standard text files. The spooling daemon will use the appropriate filters to print the data accordingly.

`-p` Use `pr` to format the files (`lpr -p` is very much like `pr | lpr`).

`-l` Print control characters and suppress page breaks.

`-t` The files contain `troff(1M)` (cat phototypesetter) binary data.

`-n` The files contain data from `ditroff` (device independent troff).

`-d` The files contain data from `tex` (DVI format from Stanford).

`-g` The files contain standard plot data as produced by the `plot` routines (see also `plot(1)` for filters used by the printer spooler).

`-v` The files contain a raster image. The printer must support an appropriate imaging model such as PostScript in order to print the image.

`-c` The files contain data produced by `cifplot`.

`-f` Interpret the first character of each line as a standard FORTRAN carriage control character.

If no *filter_option* is given (and the printer can interpret PostScript), the string "`%!`" as the first two characters of a file indicates that it contains PostScript commands.

These filter options offer a standard user interface, and all options may not be available for, nor applicable to, all printers.

DIAGNOSTICS

`lpr:` *printer*: unknown printer
The `printer` was not found in the LP database. Usually this is a typing mistake; however, it may indicate that the printer does not exist on the system. Use `lptstat -p` to find the reason.

`lpr: error on opening queue to spooler`
The connection to `lpsched` on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check if the printer spooler daemon `/usr/lib/lpsched` is running.

`lpr:` *printer*: printer queue is disabled
This means the queue was turned off with

`/usr/etc/lpc disable` *printer*

to prevent `lpr` from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a privileged user with `lpc`.

`lpr: Can't send message to the LP print service`
`lpr: Can't receive message from the LP print service`
These indicate that the LP print service has been stopped. Get help from the system administrator.

`lpr: Received unexpected message from LP print service`
It is likely there is an error in this software. Get help from system administrator.

`lpr: There is no filter to convert the file content`
Use the `lpstat -p-l` command to find a printer that can handle the file type directly, or consult with your system administrator.

`lpr: cannot access the file`
Make sure file names are valid.

NOTES

`lp` is the preferred interface.

Command-line options cannot be combined into a single argument as with some other commands. The command:

`lpr -fs`

is not equivalent to

`lpr -f -s`

Placing the `-s` flag first, or writing each option as a separate argument, makes a link as expected.

`lpr -p` is not precisely equivalent to `pr | lpr`. `lpr -p` puts the current date at the top of each page, rather than the date last modified.

Fonts for `troff(1M)` and TEX reside on the printer host. It is currently not possible to use local font libraries.

`lpr` objects to printing binary files.

The `-s` option does not use symbolic links in the compatibility package. Instead, the complete path names are used. Also, the copying is avoided only for print jobs that are run from the printer host itself. Jobs added to the queue from a remote host are always copied into the spool area. That is, if the printer does not reside on the host that `lpr` is run from, the spooling system makes a copy of the file to print, and places it in the spool area of the printer host, regardless of `-s`.

FILES

`/etc/passwd`
: Personal identification

`/usr/lib/lp/lpsched`
: System V line printer spooler

`/var/spool/lp/tmp/*`
: Directories used for spooling

`/var/spool/lp/tmp/`*system*`/*-0`
: Spooler control files

`/var/spool/lp/tmp/`*system*`/*-`*N*
: Data files specified in "`*-0`" files (*N* is an integer and > 0)

SEE ALSO

`lp(1)`, `lpq(1)`, `lprm(1)`, `plot(1)`, `pr(1)`, `lpc(1M)`, `lpsched(1M)`, `troff(1M)`.

User's Reference Manual

lprm(1)
BSD Compatibility Package

NAME

`lprm -` remove jobs from the printer queue

SYNOPSIS

`/usr/ucb/lprm` [`–P`*printer*] [`–`] [*job#* ...] [*username* ...]

DESCRIPTION

`lprm` removes a job or jobs from a printer's spooling queue. Since the spool directory is protected from users, using `lprm` is normally the only method by which a user can remove a job.

Without any arguments, `lprm` deletes the job that is currently active, provided that the user who invoked `lprm` owns that job.

When the privileged user specifies a *username*, `lprm` removes all jobs belonging to that user.

You can remove a specific job by supplying its job number as an argument, which you can obtain using `lpq(1)` . For example:

```
lpq  –Phost
host is ready and printing
Rank Owner   Job     Files   Total Size
active       wendy   385     standard input  35501 bytes
lprm –Phost 385
```

`lprm` reports the names of any files it removes, and is silent if there are no applicable jobs to remove.

`lprm` Sends the request to cancel a job to the print spooler, `lpsched`.

OPTIONS

`–P`*printer*
: Specify the queue associated with a specific printer. Otherwise the value of the `PRINTER` variable in the environment is used. If this variable is unset, the queue for the default printer is used.

`–`
: Remove all jobs owned by you. If invoked by the privileged user, all jobs in the spool are removed. Job ownership is determined by the user's login name and host name on the machine where the `lpr` command was executed.

DIAGNOSTICS

`lprm:` *printer*: unknown printer
: The `printer` was not found in the System V LP database. Usually this is a typing mistake; however, it may indicate that the printer does not exist on the system. Use `lptstat –p` to find the reason.

`lprm: error on opening queue to spooler`
: The connection to `lpsched` on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check if the printer spooler daemon `/usr/lib/lpsched` is running.

`lprm: Can't send message to the LP print service`
`lprm: Can't receive message from the LP print service`
: These indicate that the LP print service has been stopped. Get help from the system administrator.

`lprm: Received unexpected message from the LP print service`
: It is likely there is an error in this software. Get help from system administrator.

`lprm: Can't cancel request`
: You are not allowed to remove another's request.

NOTES

An active job may be incorrectly identified for removal by an `lprm` command issued with no arguments.

During the interval between an `lpq(1)` command and the execution of `lprm`, the next job in queue may have become active; that job may be removed unintentionally if it is owned by you. To avoid this, supply `lprm` with the job number to remove when a critical job that you own is next in line.

Only the privileged user can remove print jobs submitted from another host.

`lp` is the preferred interface.

FILES

`/var/spool/lp/*`
Spooling directories

SEE ALSO

`cancel(1)`, `lp(1)`, `lpq(1)`, `lpr(1)`, `lpsched(1M)`.

lpstat(1)
AT&T Spooler only

NAME

`lpstat -` print information about the status of the LP printservice

SYNOPSIS

`lpstat` [*options*]

DESCRIPTION

The `lpstat` command prints information about the current status of the LP print service.

If no options are given, then `lpstat` prints the status of all the user's print requests made by `lp` [see `lp(1)`]. Any arguments that are not *options* are assumed to be *request-IDs* as returned by `lp`. The `lpstat` command prints the status of such requests. The *options* may appear in any order and may be repeated and intermixed with other arguments. Some of the keyletters below may be followed by an optional *list* that can be in one of two forms: a list of items separated from one another by a comma, or a list of items separated from one another by spaces enclosed in quotes. For example:

```
–u "user1, user2, user3"
```

Specifying `all` after any keyletter that takes *list* as an argument causes all information relevant to the keyletter to be printed. For example, the command

```
lpstat –o all
```

prints the status of all output requests.

The omission of a *list* following such key letters causes all information relevant to the key letter to be printed. For example, the command

```
lpstat –o
```

prints the status of all output requests.

OPTIONS

`–a` [*list*]
: Reports whether print destinations are accepting requests. *list* is a list of intermixed printer names and class names.

`–c` [*list*]
: Reports name of all classes and their members. *list* is a list of class names.

`–d`
: Reports the system default destination for output requests.

`–f` [*list*] [`–l`]
: Prints a verification that the forms in *list* are recognized by the LP print service. *list* is a list of forms; the default is `all`. The `–l` option will list the form descriptions.

`–o` [*list*]
: Reports the status of output requests: *list* is a list of intermixed printer names, class names, and *request-IDs*. The keyletter `–o` may be omitted.

`–p` [*list*] [`–D`] [`–l`]
: Reports the status of printers. *list* is a list of printer names. If the `–D` option is given, a brief description is printed for each printer in *list*. If the `–l` option is given, and the printer is on the local machine, a full description of each printer's configuration is given, including the form mounted, the acceptable content and printer types, a printer description, the interface used, and so on. If the `–l` option is given and the printer is remote, the only information given is the remote machine and printer names, and the shell-commands used for file transfer and remote execution.

`–r`
: Reports whether the LP request scheduler is on or off.

`-R` Reports a number showing the position of the job in the print queue.

`-s` Displays a status summary, including the status of the LP scheduler, the system default destination, a list of class names and their members, a list of printers and their associated devices, a list of the machines sharing print services, a list of all forms currently mounted, and a list of all recognized character sets and print wheels.

`-S` [*list*] [`-l`]
Prints a verification that the character sets or the print wheels specified in *list* are recognized by the LP print service. Items in *list* can be character sets or print wheels; the default for the list is `all`. If the `-l` option is given, each line is appended by a list of printers that can handle the print wheel or character set. The list also shows whether the print wheel or character set is mounted or specifies the built-in character set into which it maps.

`-t` Displays all status information: all the information obtained with the `-s` option, plus the acceptance and idle/busy status of all printers.

`-u` [*login-ID-list*]
Displays the status of output requests for users. The *login-ID-list* argument may include any or all of the following constructs:

login-ID
auser on any system

system_name`!`*login-ID*
auser on system *system_name*

system_name`!all`
all users on system *system_name*

`all!`*login-ID*
auser on all systems

`all` all users on all systems

`-v` [*list*]
Reports the names of printers and the pathnames of the devices associated with them or remote system names for network printers: *list* is a list of printer names.

INTERNATIONALIZED ENVIRONMENT

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. `LC_TIME` governs the format and contents of date specifications in the output of status information.

If `LC_MESSAGES` or `LC_TIME` is undefined or defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all other environment variables that affect internationalization.

ENVIRONMENT VARIABLES

`TZ` Defines the time zone within date and time specifications.

FILES

```
/var/spool/lp/*
/etc/lp/*
```

SEE ALSO

`enable(1)`, `lp(1)`.

User's Reference Manual

lptest(1)
BSD Compatibility Package

NAME

`lptest -` generate lineprinter ripple pattern

SYNOPSIS

`/usr/ucb/lptest` [*length* [*count*]]

DESCRIPTION

`lptest` writes the traditional "ripple test" pattern on standard output. In 96 lines, this pattern will print all 96 printable ASCII characters in each position. While originally created to test printers, it is quite useful for testing terminals, driving terminal ports for debugging purposes, or any other task where a quick supply of random data is needed.

The *length* argument specifies the output line length if the default length of 79 is inappropriate.

The *count* argument specifies the number of output lines to be generated if the default count of 200 is inappropriate.

NOTES

If *count* is to be specified, *length* must be also be specified.

User's Reference Manual

ls(1)

NAME

`ls, lc -` list contents of directory

SYNOPSIS

`ls` [*option ...*] [*file ...*]

`/sbin/lc` [*option ...*] [*file ...*]

DESCRIPTION

`ls` is used to obtain detailed information on files and directories. The type, scope and output format of the displayed information can be specified by setting various options.

The `lc` command corresponds to `ls –Cm` (see **OPTIONS**); though all other options can also be specified.

OPTIONS

No argument specified:

`ls` lists the names of files and directories in the current working directory. The output is sorted in alphabetical order.

No option specified:

file is a directory:

`ls` lists the names of the files and directories in it. The output is sorted in alphabetical order. The only files not listed are those which begin with a dot, such as `.profile`. The output for `ls` is divided into a number of columns for terminals (as described for `–C`), otherwise it is in a single column.

file is a file:

`ls` displays the name of the file. You can use a partially qualified file name. If an asterisk (`*`) is specified for *file*, all files and directories (and their contents) are listed.

`–a` Lists all file names, including those that begin with a dot.

`–b` Non-printing characters in file names are represented in the octal notation *ddd*, where *ddd* is the octal value of the character.

`–c` Together with the `–l` option:

Lists time of last modification of the inode (file created, mode change, etc.) instead of the time at which the file was last modified.

Together with the `–t` option:

`ls` uses the time of last modification of the inode as the sort criterion instead of the time of last modification of the file.

The `–c` and `–u` options are conflicting. If a number of these options are specified, the last option specified determines the output format.

`–C` Multi-column output with entries sorted down the columns in alphabetical order.

`ls` uses the `COLUMNS` environment variable to determine the number of character positions per line. If this variable is not set, the value of the `TERM` environment variable is checked in the terminfo database in order to determine the number of columns. If this information cannot be obtained, 80 characters per line are assumed.

The `–C` and `–l` or `–1` options are conflicting. If a number of these options are specified, the last option specified determines the output format.

`–d` If *file* is a directory, `ls` lists its name (not its contents). This option is often used together with `–l` in order to obtain information on the status of the directory. If *file* is omitted, `ls` outputs a period for the current directory.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

`-e` When used with `-l`, displays extent attribute information for files with reserved space or fixed extent sizes. The extent attribute information appears as follows:

```
:res 36 ext 3 align noextend
```

This output line indicates a file with 36 blocks of reservation, a fixed extent size of 3 blocks, all extents aligned to 3 block boundaries, and the file unable to be grown once the current reservation is exhausted.

`-f` Forces each argument to be interpreted as a directory and lists the entries of each directory in the order in which they appear. If no argument is specified, the entries in the current directory are listed. This option turns on the `-a` option and turns off the `-l`, `-t`, `-F`, `-s`, and `-r` options.

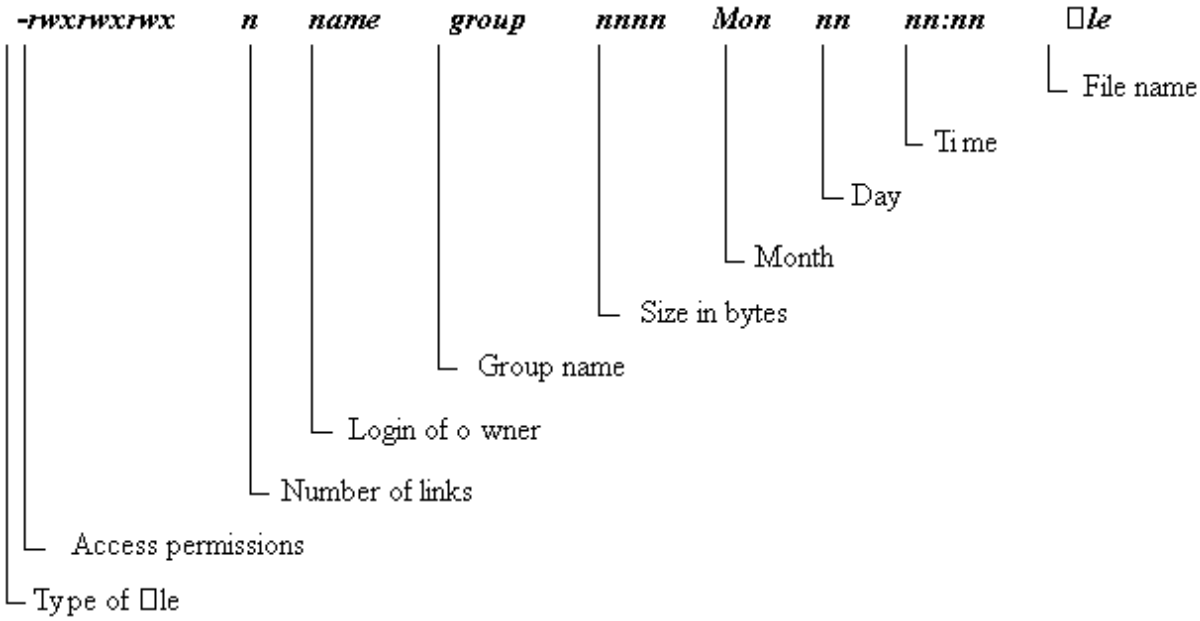
`-F` Puts a slash (`/`) after each file name if the file is a directory, an asterisk (`*`) if the file is declared as an executable, a commercial at symbol (`@`) if the file is a symbolic link, and a pipe symbol (|) if the file is a FIFO file.

`-g` Has the same effect as `-l`, except that the owner is not printed.

`-i` Prints the unique identification number (inode number) for each file in the first column of the report.

`-l` A list of detailed information is provided in long format for each file.

This information is displayed as shown below:



Whereby:

`d` for a directory

`l` for a symbolic link

`b` for a block special file

`c` for a character special file

`m` XENIX for a shared data (memory) file

`p` for a FIFO file (named pipe)

`s` XENIX for a semaphore

`-` for an ordinary file .RE

Access permissions

3 sets of 3 characters each provide information on the permissions of

- the file owner (characters 1 to 3),
- users belonging to the same group (characters 4 to 6),
- other users (characters 7 to 9).

Each set uses the following codes:

Position 1:
`r` for read permission or – for no read permission.

Position 2:
`w` for write permission or – for no write permission.

Position 3:
`x` for execute permission for directories

`s`	for execute permission with s bit set
`t`	for execute permission with t bit (sticky bit) set
`T`	for no execute permission with t bit (sticky bit) set
`S`	for the l bit (set-user-ID bit set, execute permission not set)
–	for no execute permission with no special bit set

If the s bit (set-user-ID mode) is set for the owner or the group and the corresponding x bit (execute permission) is also set, the `x` will be replaced by an `s`. The s bit cannot be set without the corresponding x bit.

In the case of group permissions, an `S` may occupy the position of the x bit if the l bit is set for the file, i.e. if mandatory locking has been enabled for the file. Neither the x bit nor the s bit can be set for the group in this case.

If the l bit is set for a file, a program using the `lockf()` function can lock read and write access to the file for as long as it is accessing that file [see `chmod(1)`].

For others permissions, the position of the x bit may be occupied by a `t` or `T`. These refer to the state of the sticky bit (t bit): `t` stands for set sticky bit with x bit, `T` for set sticky bit without x bit [see `chmod(1)`].

Number of links

Decimal number indicating the number of links to the file; at least 1.

Login of owner

Login name of the file owner.

Group name

Group name of the file owner. *Note*: The name is not output after the tenth character.

Size in bytes

Decimal number that indicates the file size in bytes.

If a special file is listed, the major device number and the minor device number are displayed instead of the file size.

Month, *Day*, *Time*

Indicates the date and time of the last modification to the file. If the last modification was made more than six months ago, the year is output instead of the time.

File name

Name of the file.

In the case of directories, the size of the entire directory is displayed in addition to the individual file sizes. This size is specified in 512-byte blocks (see *Example 4*).

If the file or directory is a symbolic link, the file name is printed followed by an arrow –> and the pathname of the referenced file.

The `–l` and `–C`, `–m` or `–x` options are conflicting. If a number of these options are specified, the last

option specified determines the output format.

`-L` With symbolic links, the link name is listed rather than the name of the original file or directory.

Example:

```
$ ls -l
-rw------- 1 hugo other 7593  Nov 30 10:48 file
-rw------- 1 hugo other    3  Dec 17 10:53 linker -> file$ ls -lL
-rw------- 1   hugo  other  7593   Nov 30 10:48 file
-rw------- 1   hugo  other  7593   Nov 30 10:48 linker
```

`-m` Lists files across the page (stream output format), separated by commas. `ls` uses the `COLUMNS` environment variable to determine the number of character positions per line. If this variable is not set, the value of the `TERM` environment variable is checked in the terminfo database in order to determine the number of columns. If this information cannot be obtained, 80 columns are assumed per line.

The `-m` and `-l` options are conflicting. If a number of these options are specified, the last option specified determines the output format.

`-n` Same effect as `-l`, except that the user ID and the group ID are displayed instead of the login and group names.

`-o` same effect as `-l`, except that the group is not displayed.

`-p` Puts a slash (/) after each file name if the file is a directory.

`-q` Non-printing characters in file names are represented by question marks.

`-r` Reverses the sorting order.

`-R` If *file* is not specified:

All files and directories in the current directory are listed, followed by a recursive listing of all subdirectories and the files contained in them.

If *file* is specified (must be a directory):

All files and subdirectories of the specified directory are listed recursively.

`-s` Prints the size of each file in column 1 (in 512-byte blocks) in addition to the file names.

`-t` Sorts by time last modified instead of by file name (latest first).

`-u` Together with the `-t` option:

Uses the time of last access as the sort criterion.

Together with the `-l` option:

Displays the time of last access instead of the time of last modification.

The `-u` and `-c` options are conflicting. If a number of these options are specified, the last option specified determines the output format.

`-x` Multi-column output with entries sorted across instead of down the page. `ls` uses the `COLUMNS` environment variable to determine the number of character positions per line. If this variable is not set, the value of the `TERM` environment variable is checked in the terminfo database in order to determine the number of columns. If this information cannot be obtained, 80 characters per line are assumed.

The `-x` and `-l` options are conflicting. If a number of these options are specified, the last option specified determines the output format.

`-1` Prints only one entry per line of output.

The `-1` and `-C` options are conflicting. If a number of these options are specified, the last option specified determines the output format.

`--` End of the list of options. All arguments after this character are interpreted as filenames, even if they begin with `-`.

file Name of the file or directory for which you want information to be listed. You can also name more than

one file or directory.

file not specified:

The contents of the current directory will be listed.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

LC_COLLATE governs the collating sequence of the output from ls, LC_CTYPE determines which characters are defined as non-printing characters in conjunction with the –q option, and LC_TIME governs the format and content of date and time strings when the –g, –l, –n, and –o options are used.

If LC_MESSAGES, LC_COLLATE, LC_CTYPE or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized. If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Listing the contents of the current directory in long format. User's login name: hugo

```
$ ls –l
total 142
drwx--x--x 2   hugo  other     48   Dec  1 16:13 ADDRESSES
-rw------- 1   hugo  other    356   Dec 17 13:58 card_a
-rw------- 1   hugo  other  24802   Dec  1 12:13 card_b
-rw------- 1   hugo  other   7890   Nov 30 10:48 card_c
-rwx--xr-x 1   hugo  other   7253   Dec 21 13:37 card_d
-rw------- 1   hugo  other   9476   Dec 21 13:37 card_e
-rwx--x--x 1   hugo  other      0   Dec 18 13:16 card_f
-rw------- 1   hugo  other    593   Nov 30 10:48 letter1
-rw------- 1   hugo  other    837   Dec 17 10:53 letter2
-rw------- 1   hugo  other   3247   Dec 17 13:46 letter3
-rw------- 1   hugo  other   5222   Nov 30 10:48 letter4
-rw-rw-rw- 1   hugo  other   4687   Dec 21 11:15 letter5
-rw------- 1   hugo  other    228   Nov 30 10:48 letter6
-r--------
1

hugo  other    105   Dec 21 13:39 typescript
```

Example 2

Multi-column output of the current directory using a line width of 80 columns per line.

```
$ ls –C
ADDRESSES   card_c      card_f      letter3     letter6
card_a      card_d      letter1     letter4     typescript
card_b      card_e      letter2     letter5
```

Example 3

Multi-column output after defining the line width as 40 columns per line.

```
$ COLUMNS=40
$ export COLUMNS
$ ls –C
ADDRESSES   card_e      letter4
card_a      card_f      letter5
card_b      letter1     letter6
```

```
card_c      letter2     typescript□
card_d      letter3
```

Example 4

Listing the current directory in long format (option `-l`), marking directories (`-p` option) and including all files that begin with a period (`-a` option). User's login name: `sisyphus`.

```
$ ls -pla□
total 120□
drwxr-xr-x  10 sisyphus other  5720   Nov 17 08:00 ./□
drwxr-xr-x  13 root     root   3380   Nov 04 11:48 ../□
-rw-------   1 sisyphus other    79   Jul 19 14:21 .profile□
-rw-------   1 sisyphus other    14   Oct 21 08:56 .rhosts□
-rwx------   1 sisyphus other   125   May 25 10:29 begin□
drwx--x--x   2 sisyphus other  3380   Nov 09 10:30 commands/□
drwx------   2 sisyphus other  1820   Oct 11 15:35 letters/□
drwxr-----   3 sisyphus other  2340   Oct 11 15:35 lingua/□
-rw-------   1 sisyphus other  2082   Nov 08 12:29 ls.ex□
-rw-------   1 sisyphus other 11597   Nov 17 07:59 ls.rc.1□
-rw-------   1 sisyphus other  1351   Jul 19 15:14 plural□
drwx------   2 sisyphus other  3380   Oct 11 15:36 post/□
drwx------   2 sisyphus other  1560   Oct 11 15:36 pro/□
drwx--x--x□
 □
2
sisyphus other  2080   Nov 07 10:43 proc/□
drwx------   2 sisyphus other  1560   Oct 11 15:36 screens/□
drwx--x--x   2 sisyphus other  1040   Nov 15 08:23 sdg/
```

Example 5

List all files and directories (and their contents). Each directory name is followed by a colon.

```
$ ls -C *□
Mailboxes  authors   bvp.col□
lingua:□
car.cdr    engl.ger  examples  terms□
screens:□
out        pause     sinix
```

FILES

`/etc/passwd`
: Contains all the login names that have been set up.

`/etc/group`
: Contains all the groups that have been set up.

SEE ALSO

`chmod(1)`, `find(1)`, `ln(1)`.

ls(1-ucb)
BSD Compatibility Package

NAME

`ls` - list the contents of a directory

SYNOPSIS

`/usr/ucb/ls` [`–aAcCdfFgilLqrRstu1`] *filename* ...

DESCRIPTION

For each *filename* which is a directory, `ls` lists the contents of the directory; for each *filename* which is a file, `ls` repeats its name and any other information requested. By default, the output is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments are processed before directories and their contents.

Permissions Field

The mode printed under the `–l` option contains 10 characters interpreted as follows. If the first character is:

`d` entry is a directory;

`b` entry is a block-type special file;

`c` entry is a character-type special file;

`l` entry is a symbolic link;

`p` entry is a FIFO (also known as "named pipe") special file;

`s` entry is an `AF_UNIX` address family socket, or

`–` entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next refers to permissions to others in the same user-group; and the last refers to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, "execute" permission is interpreted to mean permission to search the directory. The permissions are indicated as follows:

`r` the file is readable;

`w` the file is writable;

`x` the file is executable;

`–` the indicated permission is not granted.

The group-execute permission character is given as `s` if the file has the set-group-id bit set; likewise the owner-execute permission character is given as `s` if the file has the set-user-id bit set.

The last character of the mode (normally `x` or `–`) is `true` if the 1000 bit of the mode is on. See `chmod(1)` for the meaning of this mode. The indications of set-ID and 1000 bits of the mode are capitalized (`S` and `T` respectively) if the corresponding execute permission is *not* set.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

OPTIONS

`–a` List all entries; in the absence of this option, entries whose names begin with a " `.`" are *not* listed (except for the privileged user, for whom `ls` normally prints even files that begin with a " `.`").

`–A` Same as `–a`, except that " `.`" and " `..`" are not listed.

`–c` Use time of last edit (or last mode change) for sorting or printing.

`-C` Force multi-column output, with entries sorted down the columns; for `ls`, this is the default when output is to a terminal.

`-d` If argument is a directory, list only its name (not its contents); often used with `-l` to get the status of a directory.

`-f` Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off `-l`, `-t`, `-s`, and `-r`, and turns on `-a`; the order is the order in which entries appear in the directory.

`-F` Mark directories with a trailing slash ("`/`"), executable files with a trailing asterisk ("`*`"), symbolic links with a trailing at-sign ("`@`"), and `AF_UNIX` address family sockets with a trailing equals sign ("`=`").

`-g` For `ls`, show the group ownership of the file in a long output.

`-i` For each file, print the i-node number in the first column of the report.

`-l` List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. If the file is a special file the size field will instead contain the major and minor device numbers. If the time of last modification is greater than six months ago, it is shown in the format "*month date year*"; files modified within six months show "*month date time*". If the file is a symbolic link the pathname of the linked-to file is printed preceded by "`->`".

`-L` If argument is a symbolic link, list the file or directory the link references rather than the link itself.

`-q` Display non-graphic characters in filenames as the character "`?`"; for `ls`, this is the default when output is to a terminal.

`-r` Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

`-R` Recursively list subdirectories encountered.

`-s` Give size of each file, including any indirect blocks used to map the file, in kilobytes.

`-t` Sort by time modified (latest first) instead of by name.

`-u` Use time of last access instead of last modification for sorting (with the `-t` option) and/or printing (with the `-l` option).

`-1` Force one entry per line output format; this is the default when output is not to a terminal.

NOTES

NEWLINE and TAB are considered printing characters in filenames.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "`ls -s`" is much different than "`ls -s | lpr`". On the other hand, to omit this setting would be to the detriment of old shell scripts that used `ls`.

Unprintable characters in file names may confuse the column output options.

FILES

`/etc/passwd`
Get user ID's for "`ls -l`" and "`ls -o`"

`/etc/group`
Get group ID for "`ls -g`"

mach(1-ucb) - mv(1)

User's Reference Manual

mach(1)
BSD Compatibility Package

NAME

`mach` - display the processor type of the current host

SYNOPSIS

`/usr/ucb/mach`

DESCRIPTION

The `mach` command displays the processor-type of the current host.

SEE ALSO

`arch`(1), `machid`(1), `uname`(1), `uname`(2), `sysinfo`(2).

User's Reference Manual

machid(1)

NAME

machid:
a20r, cpl, i286, i386, i486, m120, m68k, mips, pdp11, rm200, rm300, rm400, rm600, rm700, sparc, t4600, u370, u3b15, u3b2, u3b5, u3b, vax - get processor type truth value

SYNOPSIS

a20r □
cpl □
i286 □
i386 □
i486 □
m120 □
m68k □
mips □
pdp11 □
rm200 □
rm300 □
rm400 □
rm600 □
rm700 □
sparc □
t4600 □
u370 □
u3b15 □
u3b2 □
u3b5 □
u3b □
vax

DESCRIPTION

The following commands will return a true value (exit code of 0) if you are on a processor that the command name indicates.

a20r True if you are on an RM200, RM300, RM400 computer.

cpl True if you are on an RM600-xx computer.

i286 True if you are on an Intel 80286 computer.

i386 True if you are on an Intel 80386 computer.

i486 True if you are on an Intel 80486 computer.

m120 True if you are on a Pyramid single-processor MIPS computer.

m68k True if you are on a Motorola 68xxx computer.

mips True if you are on a computer with MIPS processor.

pdp11
True if you are on a PDP-11/45 or -11/70 computer.

rm200
True if you are on an RM200 computer.

rm300
True if you are on an RM300 computer.

rm400
True if you are on an RM400 computer.

`rm600`
True if you are on an RM600-xx computer.

`rm700`
True if you are on an RM600-xxx computer.

`sparc`
True if you are on a Sun computer with SPARC processor.

`t4600`
True if you are on an RM600-xx computer.

`u370` True if you are on an IBM System/370 computer.

`u3b15`
True if you are on a 3B15 computer.

`u3b2` True if you are on a 3B2 computer.

`u3b5` True if you are on a 3B5 computer.

`u3b` True if you are on a 3B20 computer.

`vax` True if you are on a VAX-11/750 or -11/780 computer.

The commands that do not apply will return a false (non-zero) value. These commands are often used within makefiles [see `make(1)`] and shell procedures [see `sh(1)`] to increase portability.

SEE ALSO

`make(1)`, `sh(1)`, `test(1)`, `true(1)`, `uname(1)`.

User's Reference Manual

mail(1)

NAME

`mail -` send or read mail

SYNOPSIS

Format 1

`mail` [`–tw`] [`–m` *message_type*] [—] *recipient ...*

Format 2

`mail` [`–ehpPqr`] [`–f` *file*]

Format 3

`mail –F` *recipient ...*

Format 4

`mail –T` *mailsurr_file_recipient ...*

Format 5

`mail` [`–x`*debug_level*] [*other_mail_options ...*] *recipient ...*

DESCRIPTION

This description is divided into the following sections:

- Introduction
- Description of individual formats

  Format 1: Send mode

  Functionality in send mode

  Format 2: Read mode

  `mail` commands in read mode

  Functionality in read mode

  Format 3: Forwarding mode

  Functionality in forwarding mode

  Formats 4 and **5**: Debugging mode
- Exit status
- Files
- Examples
- See also

INTRODUCTION

`mail` enables you to send and receive electronic mail. If your computer is linked to a network, `mail` can also be used to exchange messages with users on remote systems.

The presence of mail, if any, is usually indicated when you log in. You are also notified if new mail arrives while you are using `mail`.

Messages are in the form of ASCII text. They have a message header holding information needed for message forwarding; the nature of this information is partly implementation-specific. The header is followed by a blank line and then the text of the message.

`mail` allows you to

- send messages (send mode, **Format 1**)
- read messages (read mode, **Format 2**)
- forward messages (forwarding mode, **Format 3**)

- test the sending mechanism (debugging mode, **Formats 4** and **5**)

DESCRIPTION OF INDIVIDUAL FORMATS

.SH "Format 1: Send mode"

`mail` [`–tw`] [`–m` *message_type*] [`––`] *recipient* ...

`–t` Causes a `To:` *recipient* line to be added to the start of the message header for each of the intended recipients.

`–w` Causes a letter to be sent to a remote recipient in the network without waiting for completion of the remote transfer program.

`–m` *message_type*
Causes a `Message-Type:` line to be added to the message header with the value of *message_type*. *message_type* is any text string.

`––` If *recipient* begins with a dash (–), the end of the command line options must be marked with with ––.

recipient
is a valid login name on the local system, or a network address if the computer on a network. You may specify more than one recipient.

You will find information on network addressing in the next section.

Functionality in send mode

The message to be sent is read from standard input. An EOF (<CTRL-D>) or a line consisting of only a period (.) in the first column is interpreted as the end of the input. The letter thus read is then sent to the specified recipients, i.e. placed in their mailboxes. Each new incoming message is inserted before any existing messages in a mailbox.

Note:
When reading from a file, `mail` does not treat a period on its own at the start of a line as an end-of-text character.

Components of a letter

A letter as it reaches its recipient is composed of

- some header lines
- ablank line, and
- the message body

The header lines section of the letter consists of the following elements:

- one or more UNIX postmarks of the form:

  `From` *sender date_and_time* [`remote from`*remote_system_name*]
- one or more standardized message header lines of the form:

  keyword_name: [*printable text*]

Header lines may be continued on the following line if that line starts with white space.

Keywords and the text that follows provide more detailed information on incoming mail. Which keyword lines are included in the mail is determined by the sending mode and the type of system from which the message is sent.

Typical example of *keyword_name* and *printable text*:

`Received: from` *sender@system.domain*

`Date:` *date time*

`To:` *system*`!`*recipient*

`Content-Type:` *message_type*

`Content-Length:` *message_length*

Undeliverable letters

If a letter is found to be undeliverable, it is returned to the sender with appropriate diagnostics. If `mail` is interrupted by the `SIGINT` signal (<DEL> key) during input, the entered text is saved in the file `dead.letter`. The initial attempt to append to (or create) `dead.letter` will be in the current directory. If this fails, `dead.letter` will be appended to (or created in) the user's login directory. If the second attempt also fails, no `dead.letter` processing will be done.

Network addressing

You can also specify a network address as the recipient. The addressing method used will depend on the transport mechanisms available to the local system. Consult your system administrator for details on which addressing conventions are available on the local system. The two most prevalent addressing schemes are:

- UUCP-style addressing, where addresses are specified as follows:

  remote_system`!`*user*

  A series of system names separated by exclamation points can be used to direct a letter through an extended network, such as `sysa!sysb!sysc!user`.

- Domain-style addressing, where addresses are specified as follows:

  user`@`[`subdomain.`]*domain*

Delivery notification

A number of different message headers can be added to those automatically generated to determine the form in which notification of mail is provided. These headers must be entered before the text of the message with each keyword and its options in separate lines:

The following keywords and options are available:

`Transport-Options:` [*option* ...]

`Default-Options:` [*option* ...]

`>To:` *recipient* [*option* ...]

More than one *option* may be listed together without intervening white space; but there must be slash (/) before each option. If contradictory options are used, only the first option is recognized; the conflicting options that follow are ignored.

No options specified:
: The default is `/nodelivery/return`.

The following options are available:

`/delivery`
: Inform the sender that the message was successfully delivered to the *recipient*'s mailbox.

`/nodelivery`
: Do not inform the sender of successful deliveries.

`/return`
: Inform the sender if the delivery of mail fails and return the failed message to the sender.

`/report`
: Same as `/return` except that the original message is not returned.

`/ignore`
: The sender is not informed of unsuccessful deliveries, and the failed message is not returned.

recipient
: Name of a recipient.

Format 2: Read mode

`mail` [`–ehpPqr`] [`–f` *file*]

No option specified:
: `mail` checks the default mailbox `/var/mail/$USER`. If this file contains messages, the last message received is printed on a last-in, first-out basis. `mail` then issues a `?` prompt and waits for you to enter

one of the commands described in the section on **mail commands in read mode**.

`-e` `mail` simply checks whether there are any messages, terminating with an exit status of 0 if the user has mail. Otherwise, an exit status of 1 is returned.

`-e` must not be used together with `-p`.

`-h` (header) `mail` displays the contents of UNIX postmark lines for all messages in the mailbox on a last-in, first-out basis. The messages are numbered sequentially.

For information on UNIX postmarks, see **Functionality in send mode**.

`-p` `mail` prints all mail messages in last-in, first-out order without pausing between messages and then terminates.

`-p` must not be used together with `-e`.

`-P` `mail` prints all messages individually along with all header lines in last-in, first-out order.

`-q` Causes `mail` to terminate on receiving the `SIGINT` signal (<DEL> key).

`-q` not specified:

The `SIGINT` signal interrupts only the message that is being printed. The `mail` command `p` can be used to display the current message again.

`-r` Displays messages in the order in which they were received, i.e. in first-in, first-out order.

`-r` not specified:

Messages are displayed in last-in, first-out order.

`-f` *mailfile*

`mail` is to read messages from the designated *mailfile*.

`-f` *mailfile* not specified:

Messages are read from the default mailbox `/var/mail/$USER`.

Note:

`mail` interprets the first lines of incoming mail as header lines if they start with a string of non-blank characters including a colon (:).

If you send mail which itself begins with a string of characters including a colon, you must insert a blank line before the first line of text.

mail commands in read mode

`mail` displays a question mark as the input prompt for the commands described below. Each command must be terminated with <RETURN>.

If you use one or more of the deletion commands `d`, `dp`, `m`, `r`, `s`, `w` or `y` during a `mail` session and then undo the deletion with `u` and/or exit the `mail` session with `x`, the files scheduled for deletion will be preserved in `/var/mail/$USER`.

`?`
`*` Display an overview of available `mail` commands.

<RETURN>
`+`
`n`

(next) Print next message.

`-` Print previous message.

number

Print message with specified message *number*.

`#` Print number of current message.

`!`*shell-command*

Execute specified *shell_command*.

<CTRL-D>

Causes `mail` to exit, leaving undeleted messages in the mailbox (like `q`).

`a` (arrived) Print message that arrived during the current `mail` session.

`d`
`dp`
(delete/print) Delete message and print next.

`d`*n* Delete message number *n*. Do not print the next message.

`dq` (delete/quit) Delete message and quit `mail`.

`h` (header) Display a window of headers around current message.

`h`*n* Display a window of headers around message number *n*.

`h a` (header all) Display headers of all messages in the user's mailbox.

`h d` (header/delete) Display headers of messages scheduled for deletion.

`m` [*recipient* ...]
Mail the current message to the named recipient(s) and then delete it.

recipient not specified:

Your own login name is assumed as the default *recipient*.

`p` Print current message again, overriding any indications of binary (i.e. unprintable) content.

`P` (Print) Override default brief mode and print current message again, displaying all header lines.

`q` Quit, leaving undeleted messages in the mailbox (same as <CTRL-D>).

`r` [*user* ...]
Reply to the sender, and other user(s), then delete the message. The reply is entered in the same manner as when sending mail.

user not specified:

The reply is sent to the sender only.

`s` [*file* ...]
Save the message in the file named *file* and delete the message. *file* is created in the current directory unless you specify an absolute path name. If *file* already exists, the new message is appended to it.

file not specified:

The default file `$HOME/mbox` is used.

`u`[*n*] Undelete message number *n*.

n not specified:

The last message read is undeleted.

`w` [*file* ...]
(write) Like `s` or `y`, except that header lines are not written to the named *file*.

file not specified:

The default file `$HOME/mbox` is used.

`x` Exit `mail`, leaving all messages unchanged in the mailbox. In other words, deleted messages are retained (see also <CTRL-D> and `q`).

`y` [*file* ...]
(yank) Same as `s`.

Functionality in read mode

`mail` prints the most recent incoming message on the standard output. Unless redefined with the `-P` option, the default mode for printing messages is to print only those header lines of immediate interest. These include the UNIX `From` and `>From` postmarks, the `From:`, `Date:`, `Subject:`, and `Content-Length:` header lines, and any recipient header lines such as `To:`, `Cc:`, or `Bcc:` (see also **Functionality in send mode** and the `mailx` command).

After the header lines, `mail` displays the body of the message, provided it contains no unprintable characters. Otherwise, it will issue a warning about the message having binary content and not display the message itself.

Messages that take up more than one screen page can only be halted with <CTRL-S> and restarted with <CTRL-Q>. You can also redirect the output to a file for further processing, or use the command `mailx`, which has a range of output control functions.

Using mail commands

After printing the first message, `mail` displays a "`?`" and waits for you to enter `mail` commands. Each command must be terminated with <RETURN>.

If you use one or more of the deletion commands `d`, `dp`, `m`, `r`, `s`, `w` or `y` during a `mail` session, the files scheduled for deletion will be preserved in `/var/mail/$USER` if

- you undo the deletion with `u` during the session, and/or
- exit the session with `x`.
- If `mail` is invoked without the `-q` option, interrupts (i.e. the `SIGINT` signal) will only interrupt the message being printed. The current message can be printed again using the `mail` command `p` in such cases.

  If the message to be read contains unprintable characters, `mail` will simply issue a warning about its binary contents. The `p` command can be used to print this message as well.

Mailbox file permissions

The functionality of `mail` can be manipulated by changing the permissions of the mailbox file with `chmod`. The default mode is 0660 (which allows neither reading not writing). The mode of the file may be changed to read-write (0666) or read-only (0664).

If the default mode 0660 is changed, the file will be preserved even when empty to perpetuate the desired permissions. The system administrator may override this file preservation by using the `DEL_EMPTY_MAILFILE` keyword in the system file `/etc/mail/mailcnfg`. The group ID of the mailbox file must be `mail` to allow new messages to be delivered, and the mailbox file must be writable by the `mail` group.

Format 3: Forwarding mode

`mail -F` *recipient ...*

`-F` Causes all incoming mail to be forwarded to the listed recipients, provided the mailbox is empty at the time `mail` is invoked.

recipient

A list up to 1024 bytes long, in which individual users are named as recipients and separated by either commas or white space. The list should be enclosed in double quotes to ensure that it is interpreted as a single operand of the `-F` option.

Forwarding mode can be terminated by entering the null string (`mail -F ""`) as *recipient*.

If the first character of any recipient name is the pipe symbol, the remainder of the line will be interpreted as a command to which the current mail message is to be piped (see the next section).

Functionality in forwarding mode

The `-F` option allows you to forward incoming mail to other users (or to yourself on another computer). Before activating this mode, you must first ensure that your mailbox is empty. When the command `mail -F`*recipient* is given, `mail` enters the line `Forward to`*recipient ...* as the first line in your mailbox and there after forwards all incoming messages to the named recipients.

An `Auto-Forwarded-From: ...` line will be added to the header of each forwarded message.

Executing commands in forwarding mode

If a recipient name in forwarding mode begins with the pipe symbol, the remainder of the line is interpreted as a command to which the current message is to be piped. The command, known as a *Personal Surrogate*, will execute in the recipient's home directory. If the message recipient cannot be found in the `/etc/passwd` file, the command will not be executed, and a non-delivery notification will be returned to the sender of the

message. For example:

```
mail –F "|echo %R>/dev/tty016"
```

The specified recipient name in this case is `|echo`, which begins with the pipe symbol. `mail` will therefore assume that a command is involved and execute it using the parameters that follow (i.e. display the return path to the message originator on terminal `/dev/tty016`).

It is also possible to forward mail to other recipients and pipe it to a command at the same time, as in

```
mail –F "carol,joe,|myvacationprog %R"
```

After the pipe symbol, escaped double quotes (\") should be used to cause strings with embedded whitespace to be considered as single arguments to the command being executed. No shell syntax or metacharacters may be used unless the command specified is `/usr/bin/sh`. For example,

```
mail –F "|/bin/sh –c \"shell_command_line\""
```

Certain keywords are allowed within the command being piped to and are textually substituted for *before* the command line is executed:

`%R` Return path to the message originator.

`%c` Value of the `Content-Type:` header line if present.

`%S` Value of the `Subject:` header line if present.

If the command being piped to exits with any non-zero value, `mail` assumes that message delivery has failed and generates a non-delivery notification to the message originator.

Forwarding loops

Message forwarding may result in a loop. Local loops (messages sent to `user1`, which are forwarded to `user2`, and from there back to `user1`, etc.) are detected immediately. Remote loops are not detected until a built-in hop count limit of 20 is exceeded. In both cases a non-delivery notification is sent to the message originator.

Mailbox file permissions

As a security precaution, `chmod s+g` is performed on `/var/mail/$USER` whenever forwarding is activated, and `chmod s-g` is performed when forwarding is removed. If the set-group-ID mode bit is not set when `mail` attempts to forward an incoming message to a command, the operation will fail and a non-delivery report with appropriate diagnostics will be sent to the message originator.

To ensure that forwarding works properly, the mailbox file should have the group ID of the `mail` group, and the group permission should be read-write.

Additional functionality using notify and vacation

The two shell commands `notify(1)` and `vacation(1)` provide added functionality. `notify` can be used for asynchronous notification of new mail; `vacation` provides an auto-answer capability for messages when the recipient will be unavailable for an extended period of time.

Formats 4 and 5: Debugging mode

`mail –T` *mailsurr_file* [–-] *recipient ...*

`mail` [–x*debug_level*] [*other_mail_options ...*] [–-] *recipient ...*

`–T` *mailsurr_file*

This option is useful for system administrators who need to modify a mail surrogate file, which is a file containing surrogate commands for mail routing and transport

mailsurr_file can be any file name. If the null string is entered instead (`–T ""`), `mail` will use the system file `/etc/mail/mailsurr`.

`mail` reads a message from standard input and traces the processing of *mailsurr_file* on standard output without actually sending the message.

recipient

Name(s) of one or more recipients on whom *mailsurr_file* is to be tested.

`-x` `mail` creates a file named `/tmp/MLDBG`*process_id* containing information relating to how `mail` processed the current message.

The information provided is more detailed than that provided by the `-T` option and is probably only useful to system administrators.

debug_level

Any positive or negative integer. The absolute value of *debug_level* controls the level of detail of the debug information. If *debug_level* is greater than zero, the debug information will be retained only if `mail` encounters problems in processing the message. If *debug_level* is less than zero, all the information that is relevant for processing will be retained. If *debug_level* is equal to zero, no debugging is performed.

debug_level overrides the value of `DEBUG` in `/etc/mail/mailcnfg`.

other_mail_options

`mail` send options to be debugged.

`--` If *recipient* begins with a dash (-), the end of the command line options must be marked with `--`.

recipient

Name(s) of one or more recipients on whom the sending mechanism is to be tested.

EXIT STATUS

`0` There were messages in the mailbox, and `mail` has been terminated successfully. If no error occurred, the exit status for mailing and debugging is equal to zero.

`1` There were no messages in the mailbox or an error occurred during initialization.

`>1` An error occurred after initialization.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. `LC_CTYPE` governs character classes and character conversion (shifting). `LC_COLLATE` governs the collating sequence.

The `LC_TIME` environment variable governs the format of date and time strings. If `LC_MESSAGES`, `LC_CTYPE`, `LC_COLLATE` or `LC_TIME` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Sending a message

User `connie` wishes to send a message to two other users: to `annette` on the same host and to `cindy` on the remote host `saturn`. A `To` header line (`-t`) is to be included for each of the recipients. In addition, the sender is to be notified whether the message was successfully delivered (`Transport-Options`).

`$ mail -t annette saturn!cindy` or `$ mail -t annette cindy@saturn`

```
Transport-Options: /delivery
Hi there!
Friday's meeting has had to be canceled, as the boss is
suffering from Repetitive Strain Injury brought on by
playing too much Tetris.
Bye for now, Connie
.
$
```

Example 2

Read and reply to a message

User `cindy` receives the message `you have mail`, so she has the new message displayed. She then sends a reply to the sender and all other recipients (`r`) and thus schedules the message for deletion. Since she subsequently quits `mail` with `q`, the message is deleted.

```
$ mail
From connie Wed Jun 19 19:36:07 1991 remote from pluto.dept3
Received: from pluto.dept3 (pluto) by saturn.dept3.cortinus.de
          at Wed, 19 Jun 91 19:36:05  +0100 (5.52.1/COR-1.1) with SMTP
From: connie@pluto.dept3
Date: Wed, 19 Jun 91 19:24 MDT
To: annette@pluto
To: cindy@saturn
Content-Type: text
Content-Length: 199
Hi there!
Friday's meeting has had to be canceled, as the boss is
suffering from Repetitive Strain Injury brought on by
playing too much Tetris.
Bye for now, Connie
? r pluto!annette
mail pluto!connie pluto!annette
Get you facts straight!
It was quite definitely Lemmings, not Tetris.
Cindy
.
q
$
```

FILES

`dead.letter`
: File that is used by `mail` to save messages that could not be mailed, e.g. due to an error or an interrupt with <DEL> during input. This file is created in the user's current or login directory and is generally appended to, not overwritten.

`$HOME/mbox`
: Default file in which messages are saved by `s`, `w` and `y` commands.

`/etc/passwd`
: System file containing all login names.

`/etc/mail/mailsurr`
: System file used for `mail` system debugging.

`/etc/mail/mailcnfg`
: File containing initialization information.

`/tmp/ma*`
: Temporary file.

`/tmp/MLDBG*`
: Debug trace file.

`/var/mail/*.lock`
: Lock for mail directory.

`/var/mail/:saved`
: Directory for holding temporary files to prevent loss of data in the event of a system crash.

`/var/mail/$USER`
: Default mailbox (see `$MAIL`) for `$USER`, i.e. the *mailfile* in which `mail` looks for incoming messages.

SEE ALSO

`chmod(1)`, `login(1)`, `mailx(1)`, `notify(1)`, `sh(1)`, `uuencode(1)`, `vacation(1)`, `write(1)`,

mail_pipe(1M), mailcnfg(4), mailsurr(4).

User's Reference Manual

mail(1-ucb)

NAME

`mail` - send and receive mail

SYNOPSIS

`/usr/ucb/mail` [`-iInv`] [`-s` *subject*] [`-c`*cc-addr*] [`-b` *bcc-addr*] *to-addr* ...

`/usr/ucb/mail` [`-iInNv`] `-f` [*name*]

`/usr/ucb/mail` [`-iInNv`] [`-u` *user*]

DESCRIPTION

`/usr/ucb/mail` is a intelligent mail processing system, which has a command syntax reminiscent of `ed`(1) with lines replaced by messages.

OPTIONS

`-i` Ignore tty interrupt signals. This is particularly useful when using `mail` on noisy phone lines.

`-I` Force `mail` to run in interactive mode even when input isn't a terminal. In particular, the `~` special character when sending mail is only active in interactive mode.

`-n` Inhibit reading `/etc/Mail.rc` upon startup.

`-N` Inhibit the initial display of message headers when reading mail or editing a mail folder.

`-v` Verbose mode. The details of delivery are displayed on the user's terminal.

`-s` *subject*
Specify the subject on command line (only the first argument after the `-s` option is used, therefore be careful to quote subjects containing spaces.)

`-c` *cc-addr*
Send carbon copies to list of users. List should be a comma-separated list of names.

`-b` *bcc-addr*
Send blind carbon copies to list of users. List should be a comma-separated list of names.

`-f` [*name*]
Read in the contents of your `mbox` (or the file specified as *name*) for processing; when you quit, `mail` writes undeleted messages back to this file.

`-u` *user*
Is equivalent to: `mail -f /var/spool/mail/user`.

Sending Mail

To send a message to one or more people, `mail` can be invoked with arguments which are the names of people to whom the mail will be sent. You are then expected to type in your message, followed by a CTRL-D at the beginning of a line. The section below **Replying to or Originating Mail**, describes some features of `mail` available to help you compose your letter.

Reading Mail

In normal usage `mail` is given no arguments and checks your mail out of the post office, then prints out a one line header of each message found. The current message is initially the first message (numbered 1) and can be printed using the `print` command [which can be abbreviated (`p`)]. You can move among the messages much as you move between lines in `ed`, with the commands `+` and `-` moving backwards and forwards, and with simple numbers.

Disposing of Mail

After examining a message you can `delete` (`d`) the message or `reply` (`r`) to it. Deletion causes `mail` to

forget about the message. This is not irreversible; You can `undelete` (`u`) the message by giving its number, or the mail session can be aborted by giving the `exit` (`x`) command. Deleted messages will, however, usually disappear never to be seen again.

Specifying Messages

Commands such as `print` and `delete` can be given a list of message numbers as arguments to apply to a number of messages at once. Thus `delete 1 2` deletes messages 1 and 2, while `delete 1-5` deletes messages 1 through 5. The special name `*` addresses all messages, and `$` addresses the last message; thus the command `top` which prints the first few lines of a message could be used in `top *` to print the first few lines of all messages.

Replying to or Originating Mail

You can use the `reply` command to set up a response to a message, sending it back to the person that it was from. Text you then type in, up to an end-of-file, defines the contents of the message. While you are composing a message, `mail` treats lines beginning with the character `~` specially. For instance, typing `~m` (alone on a line) will place a copy of the current message into the response, right shifting it by a tabstop (see `indentprefix` variable, below). Other escapes will set up subject fields, add and delete recipients to the message and allow you to escape to an editor to revise the message or to a shell to run some commands. (These options are given in the summary below.)

Ending a Mail Processing Session

You can end a `mail` session with the `quit` (`q`) command. Messages which have been examined go to your `mbox` file unless they have been deleted, in which case they are discarded. Unexamined messages go back to the post office. (See the `-f` option above).

Personal and Systemwide Distribution Lists

It is also possible to create a personal distribution lists so that, for instance, you can send mail to `cohorts` and have it go to a group of people. Such lists can be defined by placing a line like

```
alias cohorts bill ozalp jkf mark kridle@ucbcory
```

in the file `.mailrc` in your home directory. The current list of such aliases can be displayed with the `alias` command in `mail`. System wide distribution lists can be created by editing `/etc/aliases` [see `aliases(4)` and `sendmail(1M)`], and have a different syntax. Personal aliases will be expanded in mail sent to others so that they will be able to reply to the recipients. Systemwide aliases are not expanded when the mail is sent, but any reply returned to the machine will have the systemwide alias expanded, as all mail goes through `sendmail`.

Mail has a number of options which can be set in the `.mailrc` file to alter its behavior; thus `Set askcc` enables the `askcc` feature. These options are summarized below.

Mail Commands

Each command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety – the first command which matches the typed prefix is used. For commands which take message lists as arguments, if no message list is given, then the next message which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, `mail` types: `No applicable messages` and aborts the command.

`-` Print out the preceding message. If given a numeric argument *n*, goes to the *n*th previous message and prints it.

`?` Print a brief summary of commands.

`!` Execute the shell [see `sh(1)` and `csh(1)`] command which follows.

`Print` or `P`
Like `print` but also prints out ignored header fields. See also `print`, `ignore` and `retain`.

`Reply` or `R`
: Reply to originator. Do not reply to other recipients of the original message.

`Type` or `T`
: Identical to the `Print` command.

`alias` or `a`
: With no arguments, print out all currently-defined aliases. With one argument, print out that alias. With more than one argument, create a new alias or changes an old one.

`alternates` or `alt`
: The `alternates` command is useful if you have accounts on several machines. It can be used to inform `mail` that the listed addresses are really you. When you reply to messages, `mail` will not send a copy of the message to any of the addresses listed on the alternates list. If the `alternates` command is given with no argument, the current set of alternate names is displayed.

`chdir` or `c`
: Change the user's working directory to that specified, if given. If no directory is given, then change to the user's login directory.

`copy` or `co`
: The `copy` command does the same thing that `save` does, except that it does not mark the messages for deletion when you quit.

`delete` or `d`
: Take a list of messages as an argument and mark them all as deleted. Deleted messages will not be saved in `mbox`, nor will they be available for most other commands.

`dp` or `dt`
: Delete the current message and print the next message. If there is no next message, `mail` types: `at EOF`.

`edit` or `e`
: Take a list of messages and point the text editor at each one in turn. On return from the editor, the message is read back in.

`exit`, `ex` or `x`
: Effect an immediate return to the shell without modifying the user's system mailbox, `mbox` file or edit file in `-f`.

`file` or `fi`
: The same as `folder`.

`folders`
: List the names of the folders in your folder directory.

`folder` or `fo`
: The `folder` command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read in the new file. Some special conventions are recognized for the name: `#` means the previous file, `%` means your system mailbox, `%`*user* means *user*'s system mailbox, `&` means your `mbox` file, and `+`*folder* means a file in your *folder* directory.

`from` or `f`
: Take a list of messages and print their message headers.

`headers` or `h`
: List the current range of headers, which is an 18-message group. If a `+` argument is given, then the next 18-message group is printed, and if a `-` argument is given, the previous 18-message group is printed.

`help` A synonym for `?`.

`hold`, `ho` or `preserve`
: Take a message list and mark each message therein to be saved in the user's system mailbox instead of in `mbox`. Does not override the `delete` command.

`ignore`
Add the list of header fields named to the ignored list. Header fields in the ignored list are not printed on your terminal when you print a message. This command is very handy for suppression of certain machine-generated header fields. The `Type` and `Print` commands can be used to print a message in its entirety, including ignored fields. If `ignore` is executed with no arguments, it lists the current set of ignored fields.

`mail` or `m`
Take as argument login names and distribution group names and send mail to those people.

`mbox` Indicate that a list of messages be sent to `mbox` in your home directory when you quit. This is the default action for messages if you do not have the `hold` option set.

`next` or `n`
Like `+` or RETURN. Go to the next message in sequence and type it. With an argument list, type the next matching message.

`preserve` or `pre`
A synonym for `hold`.

`print` or `p`
Take a message list and type out each message on the user's terminal.

`quit` or `q`
Terminate the session, saving all undeleted, unsaved messages in the user's `mbox` file in his/her login directory, preserving all messages marked with `hold` or `preserve` or never referenced in his/her system mailbox, and removing all other messages from his/her system mailbox. If new mail has arrived during the session, the message: `You have new mail` is output. If output while editing a mailbox file with the `-f` option, the edit file is rewritten. A return to the shell is performed, unless the rewrite of the edit file fails, in which case the user can quit with the `exit` command.

`reply` or `r`
Take a message list and send mail to the sender and all recipients of the specified message. The default message must not be deleted.

`respond`
A synonym for `reply`.

`retain`
Add the list of header fields named to the retained list. Only the header fields in the retained list are shown on your terminal when you print a message. All other header fields are suppressed. The `Type` and `Print` commands can be used to print a message in its entirety. If `retain` is executed with no arguments, it lists the current set of retained fields.

`save` or `s`
Take a message list and a filename and append each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal.

`set` or `se`
With no arguments, print all variable values. Otherwise, set option. Arguments are of the form *option=value* (no space before or after `=`) or *option*. Quotation marks may be placed around any part of the assignment statement to quote blanks or tabs, for example, "`set indentprefix=->`".

`saveignore`
`saveignore` is to `save` what `ignore` is to `print` and `type`. Header fields thus marked are filtered out when saving a message by `save` or when automatically saving to `mbox`.

`saveretain`
`saveretain` is to `save` what `retain` is to `print` and `type`. Header fields thus marked are the only ones saved with a message when saving by `save` or when automatically saving to `mbox`. `saveretain` overrides `saveignore`.

`shell` or `sh`
Invoke an interactive version of the shell.

`size` Take a message list and print out the size in characters of each message.

`source`
The `source` command reads commands from a file.

`top` Take a message list and print the top few lines of each. The number of lines printed is controlled by the `toplines` variable and defaults to five.

`type` or `t`
A synonym for `print`.

`unalias`
Take a list of names defined by `alias` commands and discard the remembered groups of users. The group names no longer have any significance.

`undelete` or `u`
Take a message list and mark each message as not being deleted.

`unread` or `U`
Take a message list and mark each message as not having been read.

`unset`
Take a list of option names and discard their remembered values; the inverse of `set`.

`visual` or `v`
Take a message list and invoke the display editor on each message.

`write` or `w`
Similar to `save`, except that *only* the message body (without the header) is saved. Extremely useful for such tasks as sending and receiving source program text over the message system.

`xit` or `x`
A synonym for `exit`.

`z` `mail` presents message headers in windowfuls as described under the `headers` command. You can move `mail`'s attention forward to the next window with the `z` command. Also, you can move to the previous window by using `z-`.

Tilde Escapes

Here is a summary of the tilde escapes, which are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the `escape` option.

`~!` *command*
Execute the indicated shell command, then return to the message.

`~b` *name ...*
Add the given names to the list of carbon copy recipients but do not make the names visible in the "Cc:" line ("blind" carbon copy).

`~c` *name ...*
Add the given names to the list of carbon copy recipients.

`~d` Read the `dead.letter` file from your home directory into the message.

`~e` Invoke the text editor for the messages collected so far. When the editing session is finished, you may continue appending text to the message.

`~f` *messages*
Read the named messages into the message being sent. If no messages are specified, read in the current message. Message headers currently being ignored (by the `ignore` or `retain` command) are not included.

`~F` *messages*
Identical to `~f`, except all message headers are included.

`~h` Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.

~m *messages*
Read the named messages into the message being sent, indented by a tab or by the value of `indentprefix`. If no messages are specified, read the current message. Message headers currently being ignored (by the `ignore` or `retain` command) are not included.

~M *messages*
Identical to `~m`, except all message headers are included.

~p Print out the messages collected so far, prefaced by the message header fields.

~q Abort the message being sent, copying the message to `dead.letter` in your home directory if `save` is set.

~r *filename*
Read the named file into the message.

~s *string*
Force *string* to become the current subject field.

~t *names ...*
Add the given *names* to the direct recipient list.

~v Invoke an alternate editor (defined by the `VISUAL` option) for the messages collected so far. Usually, the alternate editor will be a screen editor. After you quit the editor, you may resume appending text to the end of your message.

~w *filename*
Write the message onto the named file.

~| *command*
Pipe the message through *command* as a filter. If *command* gives no output or terminates abnormally, retain the original text of the message. The command `fmt(1)` is often used as *command* to rejustify the message.

~: *mail-command*
Execute the given *mail-command*. Not all commands, however, are allowed.

~~ *string*
Insert the string of text in the message prefaced by a single `~`. If you have changed the escape character, then you should double that character in order to send it.

Mail Options

Options are controlled via `set` and `unset` commands. Options may be either binary, in which case it is only significant to see whether they are set or not; or string, in which case the actual value is of interest. These options can be set in `/etc/Mail.rc` for the system, or locally in the user's `.mailrc` file.

The binary options include the following:

`append`
Force messages saved in `mbox` to be appended to the end rather than prepended. This should always be set (perhaps in `/etc/Mail.rc`).

`ask` Force `mail` to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

`askcc`
Force you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.

`autoprint`
Force the `delete` command to behave like `dp` – thus, after deleting a message, the next one will be typed automatically.

`debug`
Setting the binary option debug is the same as specifying `–d` on the command line and causes `mail` to output all sorts of information useful for debugging `mail`.

`dot` Force `mail` to interpret a single period on a line as the terminator of a message you are sending.

`hold` This option is used to hold messages in the system mailbox by default.

`ignore`
Force interrupt signals from your terminal to be ignored and echoed as @'s.

`ignoreeof`
An option related to `dot` is `ignoreeof` which makes `mail` refuse to accept a CTRL-D as the end of a message. `ignoreeof` also applies to `mail` command mode.

`metoo`
Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.

`noheader`
Setting the `noheader` option is the same as giving the `–N` option on the command line.

`nosave`
Normally, when you abort a message (by pressing CTRL-C or DEL twice) `mail` copies the partial letter to the file `dead.letter` in your home directory. Setting the binary `nosave` option prevents this.

`Replyall`
Reverse the sense of `reply` and `Reply` commands.

`quiet`
Suppress printing of the version when first invoked.

`verbose`
Setting the `verbose` option is the same as using the `–v` option on the command line. When `mail` runs in verbose mode, the actual delivery of messages is displayed on the user's terminal.

The following options have string values:

`EDITOR`
Pathname of the text editor to use in the `edit` command and `~e` escape. If not defined, a default editor is used.

`LISTER`
Pathname of the directory lister to use in the `folders` command. Default is `/bin/ls`.

`PAGER`
Pathname of the program to use in the `more` command or when the `crt` variable is set. The default paginator `more(1)` is used if this option is not defined.

`SHELL`
Pathname of the shell to use in the `!` command and the `~!` escape. A default shell is used if this option is not defined.

`VISUAL`
Pathname of the text editor to use in the `visual` command and `~v` escape.

`crt` The `crt` option is used as a threshold to determine how long a message must be before `PAGER` is used to read it. If `crt` is set without a value, the height of the terminal screen stored in the system is used to compute the threshold [see `stty(1)`].

`escape`
If defined, the first character of this option gives the character to use in place of `~` to denote escapes.

`folder`
The name of the directory to use for storing folders of messages. If this name begins with a `/`, `mail` considers it to be an absolute pathname; otherwise, the `folder` directory is found relative to your home directory.

`MBOX` The name of the mbox file. It can be the name of a folder. The default is `mbox` in the user's home directory.

`record`
If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not saved.

`indentprefix`
: String used by the `~m` escape for indenting messages, in place of the normal tab character (`^I`). Be sure to quote the value if it contains spaces or tabs.

`toplines`
: If defined, gives the number of lines of a message to be printed out with the `top` command; normally, the first five lines are printed.

BUGS

There are some options that are not documented here. Most are not useful to the general user.

Usually, `/usr/ucb/mail` is just a link to `/usr/ucb/Mail`, which can be confusing.

ENVIRONMENT

`mail` utilizes the `HOME` and `USER` environment variables.

FILES

`/var/spool/mail/*`
: Post office

`~/mbox`
: User's old mail

`~/.mailrc`
: File giving initial mail commands

`/tmp/R*`
: Temporary files

`/usr/share/lib/mail/Mail.help*`
: Help files

`/etc/Mail.rc`
: System initialization file

SEE ALSO

`fmt`(1), `vacation`(1), `newaliases`(1M), `sendmail`(1M), `aliases`(4).

User's Reference Manual

mailalias(1)

NAME

`mailalias -` translate mail alias names

SYNOPSIS

`mailalias` [*option ...*] *name ...*

DESCRIPTION

Since mail addresses for electronic messages are often long and cumbersome, simple alias names may be defined for complex addresses.

The `mailalias` command helps to determine the original mail addresses which correspond to simplified aliases.

`mailalias` is called by `mail`.

OPTIONS

No option specified:
A list of mail addresses corresponding to the specified alias *name* is written to standard output.

`-s` (silent) Normally, if more than one name is being translated, each line of output will be prefixed with the *name* being translated. The `-s` option causes this prefix to be omitted.

`-v` (verbose) Causes debugging information to be written to standard output. This allows the user to trace program execution.

name One or more mail aliases.

Mode of operation

`mailalias` finds mail addresses by performing the following steps:

- It looks for the file `/var/mail/name`. If it finds it, it prints *name* and exits.
- It looks for matching mail addresses in the user's local alias file `$HOME/lib/names`. If it finds a line beginning with the word *name*, it writes the rest of the line to standard output and exits.
- It looks for matching mail addresses in the system-wide alias files, which are listed in `/etc/mail/namefiles`. If it finds a line beginning with the word *name*, it writes the rest of the line to standard output and exits.

  By default, the system file `/etc/mail/namefiles` lists the directory `/etc/mail/lists` and the file `/etc/mail/names`.

  If the specified alias file is a directory name *dir*, `mailalias` looks for a file *dir*`/`*name*.
- If no mail addresses are found in any of the preceding steps (for example, if you have mistyped the alias or there is no such alias), `mailalias` simply prints the alias *name* and exits.

Format of alias files

The alias files contain lines in the following format:

```
alias_name list-of-addresses
```

Individual addresses on these lines are separated by white space. Lines may be continued by placing a backslash (\) at the end of the line. Lines beginning with a hash character (#) are comments.

EXAMPLES

Example 1

Finding the original mail address for the alias `zaphod`:

```
$ mailalias zaphod
```

```
phil@ap343steve
```

Example 2

Finding the original mail address for the alias `zaphod`, with program execution being traced on the screen:

```
$ mailalias -v zaphod
translate_name(zaphod, /etc/mail/lists /etc/mail/names , )
lookup(zaphod, /home3/gast/lib/names, )
lookup(zaphod, /etc/mail/lists, )
lookup(zaphod, /etc/mail/names, )
phil@ap343steve
```

FILES

`$HOME/lib/names`
: File containing private aliases.

`/etc/mail/namefiles`
: List of files to be searched.

`/etc/mail/names`
: Standard file to be searched.

SEE ALSO

`mail`(1), `uucp`(1), `smtp`(1M), `smtpd`(1M), `smtpqer`(1M), `smtpsched`(1M), `tosmtp`(1M).

User's Reference Manual

mailx(1)

NAME

`mailx` - interactive message processing system (mail extended)

SYNOPSIS

Format 1

`mailx` [*option* ...]

Format 2

`mailx` [*option* ...] *recipient* ...

DESCRIPTION

This description is divided into the following sections:

- Introduction
- Description of formats

  Format 1: Read mode

  `mailx` commands in read mode

  Input format

  Functional overview

  Descriptions in alphabetical order

  Functionality in read mode

  Format 2: Send mode

  `mailx` commands in send mode (tilde commands)

  Input format

  Functional overview

  Descriptions in alphabetical order

  Functionality in send mode
- `mailx` command and startup files
- Variables

  `mailx` variables

  Environment variables
- Note on error messages
- Files
- Examples
- See also

INTRODUCTION

`mailx` enables you to send and receive electronic mail. If your computer is linked to a network, `mailx` can also be used to exchange messages with users on remote systems.

The presence of mail, if any, is usually indicated when you log in. You are also notified if new mail arrives while you are using `mailx`.

Messages are in the form of ASCII text. They have a message header holding information needed for message forwarding; the nature of this information is partly implementation-specific. The header is followed by a blank line and then the text of the message.

`mailx` is an extended version of the `mail` utility. It allows you to:

- check for the presence of mail (read mode, **Format 1**, option –e)
- read messages (read mode, **Format 1**)
- use Reliant UNIX commands to process incoming messages (read mode, **Format 1**, mailx commands !, | and pipe)
- send messages (send mode, **Format 2**)
- use an editor to edit messages during a mailx session (read and send mode, mailx commands edit, visual, ~e and ~v).

Unlike mail, mailx automatically collects read messages in a user-specific mailbox (HOME/mbox by default).

DESCRIPTION OF FORMATS

Format 1: Read mode

mailx [*option* ...]

No option specified:
mailx behaves as described in **Functionality in read mode**.

–e mailx simply checks whether there are any messages, terminating with an exit status of 0 if the user has mail. Otherwise, an exit status of 1 is returned.

mailx does not execute any startup files (see **mailx command and startup files**).

–f [*filename*]
mailx reads messages from *filename*.

file not specified:

mailx reads messages from the user-specific mailbox $HOME/mbox.

–f *file* not specified:

mailx reads messages from the default mailbox /var/mail/$USER.

–F (F – file) mailx records all messages which are sent to a log file. The file name is the name of the first specified recipient. It will be stored in your HOME directory and can be used with mailx like a mailbox.

–F not specified:
mailx looks for the file name of the log file in the mailx variable record. If the variable is not set, no log file will be generated.

–H mailx displays only the header summaries and terminates. The exit status is 0 if the user has mail, otherwise 1.

The structure of a header summary is described in **Functionality in read mode**).

–i (i – ignore) mailx ignores the signal SIGINT (see the mailx variable ignore).

–I Only allowed in conjunction with –f.

mailx includes the newsgroup and article-id header lines when printing mail messages. Only useful for "News" system users.

–n mailx does not initialize from the global startup file /etc/mail/mailx.rc (see **mailx command and startup files**).

–N mailx does not print the initial header summary.

–T *filename*
mailx records message-id and article-id header lines in *filename* once the messages have been read.

This option automatically sets the –I option.

–u *login_name*
mailx reads messages from the default mailbox of the specified user, provided you have the required read permission.

`-V` `mailx` displays its version number and exits.

`-~` Permit tilde escapes no matter where the input is coming from.

mailx commands in read mode

Input format

`mailx` commands in read mode have the following format:

[*command*] [*msglist*] [*argument* ...]

command

Name of a `mailx` command. Most command names can be abbreviated. In **Descriptions in alphabetical order** the accepted abbreviations are shown in bold print.

command not specified:

If you simply press <RETURN> at the `mailx` prompt, `mailx` executes the `print` command.

msglist

One or more messages to be processed by the command. If several messages are specified, they are separated by blanks.

msglist can be one or more of the following:

n Message number *n*

`.` The current message (marked > in the header summary).

`^` The first undeleted message.

`$` The last message.

`*` All messages.

`+` The next message.

`-` The preceding message.

n–*m* Messages *n* through *m* inclusive.

login_name

All messages from the specified user.

/str All messages with the string *str* in the subject field (the case is ignored).

`:`*msgtype*

All messages of type *msgtype*, which can be any of the following:

`d` deleted messages

`n` new messages

`o` old messages

`r` read messages

`u` unread messages

msglist not specified:

`mailx` defaults to the current message.

argument

String argument as appropriate to the command (see descriptions). If *argument* is a file name, the usual shell metanotation may be used. If a string containing blanks is to be interpreted as a single argument, it must be enclosed in double quotes.

Functional overview

This section provides an overview of all `mailx` read-mode commands, grouped by function. Some commands may appear more than once.

The overview is followed by descriptions of all the commands in alphabetical order.

Most commands can be abbreviated. In **Descriptions in alphabetical order** the abbreviated forms are shown in bold print.

Help functions

`?` Display summary of `mailx` commands

`help` Display summary of `mailx` commands

`list` List names of all `mailx` commands

`=` Display current message number

`size` Display size of message

`from` Display header summary

`z+` Scroll header display one page forward

`z-` Scroll header display one page back

`headers`
Display screen page of headers

`top` Display first five lines of message header

`folders`
List contents of directory defined by `mailx` variable `folder`

`version`
Display `mailx` version number

Quitting mailx

`exit` Quit `mailx` without changing mailbox

`xit` Quit `mailx` without changing mailbox

`quit` Quit `mailx`

Displaying the header summary

`from` Display header summary

`headers`
Display screen page of headers

`z+` Scroll header display one page forward

`z-` Scroll header display one page back

Manipulating and displaying the message header

`discard`
Suppress message header fields

`undiscard`
Undo the effect of `discard`

`ignore`
Suppress message header fields

`unignore`
Undo the effect of `ignore`

`top` Display first five lines of message header

`retain`
Display only specified fields of message header

Displaying messages

`print`
Display message

`type` Display message

`next` Skip to next matching message

`Print`

Display message with full message header, ignoring `discard`

`Type` Display message with full message header, ignoring `discard`

Editing messages

`edit` Call editor to edit message (value of variable `EDITOR`, default: `ed`)

`visual`
Call editor to edit message (value of variable `VISUAL`, default: `vi`)

Changing the mailbox

`file` Close current mailbox and open named one

`folder`
Close current mailbox and open named one

Saving messages

`hold` Hold messages in mailbox

`preserve`
Hold messages in mailbox

`save` Write message to file

`copy` Write message to file

`write`
Write message to file, omitting header

`mbox` Write message to user's standard mailbox

`touch`
Write message to user's standard mailbox

Deleting messages

`delete`
Delete message

`dp` Delete message, display next

`dt` Delete message, display next

Switch to send mode and send or reply to message

`mail` Send message

`Mail` Send message, recording copy in file

`reply`
Reply to message

`respond`
Reply to message

`followup`
Reply to message, recording reply

`Reply`
Reply to list of messages

`Respond`
Reply to list of messages

`Followup`
Reply to list of messages, recording replies

Undoing mailx commands during a mailx session

`undelete`
Restore deleted messages

`touch`
Undo effect of `hold`

`hold` Undo effect of `touch`

`undiscard`
Undo effect of `discard`

`unignore`
Undo effect of `ignore`

`unset`
Erase variables

`unalias`
Erase aliases

Invoking command interpreters, running shell commands

`!` Escape to shell

`!!` Repeat last shell command

`shell`
Invoke command interpreter

`pipe` Pipe messages to standard input of shell command

`|` Pipe messages to standard input of shell command

Miscellaneous

`#` Null command (for including comments in command files)

`=` Display current message number

`alias`
Declare aliases for mail recipients (same as `group`)

`alternates`
Declare alternate names for your login name

`cd` Change directory

`chdir`
Change directory

`echo` Echo string (like Reliant UNIX `echo` command)

`folders`
List contents of directory defined by `mailx` variable `folder`

`group`
Declare aliases for mail recipients (same as `alias`)

`if` *mode cmdlist1* `else` *cmdlist2* `endif`
if construct which selects a command list to execute on the basis of the mode (send, read)

`set` Set variables

`size` Display size of message

`source`
Read and execute command file

`unset`
Erase variables

mailx commands not allowed in command files

`!`
Escape to shell

`edit` Call editor to edit message

`followup`
Reply to message, recording reply

`Followup`
Reply to list of messages, recording replies

`mail` Send message

`Mail` Send message, recording copy in file

`reply`
Reply to message

`Reply`
Reply to list of messages

`respond`
Reply to message

`Respond`
Reply to list of messages

`shell`
Invoke command interpreter

`visual`
Call editor to edit message

Descriptions in alphabetical order

Some command names have synonyms. The full description is always next to the name which comes first in alphabetical order.

Most of these commands can be used both interactively and in command files. Exceptions to this rule are mentioned at the appropriate place (see **Functional overview**).

The bold print in the command names refers to the abbreviated forms.

`!`*shell_command*
Executes *shell_command*. By default the command interpreter specified in the `mailx` variable `SHELL` will be invoked and the specified command line will be passed to it.

If `SHELL` is not set, `/bin/sh` will be invoked.

If the `bang` variable is set, the last shell command executed is saved by `mailx` and can be repeated with `!!`.

The `!` command is not permitted in a command file.

#*comment*
This is the null command used to introduce comments in command files (e.g. `.mailrc`).

`=` Displays the current message number.

`?` Displays a summary of all `mailx` commands.

a`lias` [*alias-name* [*recipient* ...]]
g`roup` [*alias-name* [*recipient* ...]]
Declares an alias for the given recipients. The defined recipients are substituted when you use the aliases as recipients.

alias-name
Any string.

alias-name not specified:

`mailx` displays a list of defined aliases.

recipient not specified:

`mailx` displays the definitions for *alias-name*.

alt`ernates` [*name* ...]
Declares alternate names for your login name. Once you reply to a message, `mailx` deletes these alternative names from the list of recipients.

name

String for the alternate name.

name not specified:

`mailx` displays the current list of alternate names.

cd [*directory*]
ch`dir` [*directory*]
Changes to the indicated directory.

directory not specified:

`mailx` changes to `$HOME`.

c`opy`[[*msglist*] *file*]
Copies the specified messages to the named *file*. If *file* already exists, it is extended.

The messages are marked as read (`O`) and copied to the user-specific mailbox when the default mailbox is closed.

No argument specified:

`mailx` appends the current message to `$HOME/mbox`. If you do not subsequently delete the message, it will be saved again the next time the default mailbox is closed.

It is possible to include a `copy` command in a command file, but then any subsequent commands which are intended to operate on a message list will be ignored.

C`opy` [*msglist*]
Copies the specified messages to a file in the current directory. The name of this file is derived from the name of the author of the first message in the message list (From entry).

If this file already exists, it is extended.

The messages are marked as read (`O`) and copied to the user-specific mailbox when the default mailbox is closed.

It is possible to include a `Copy` command in a command file, but then any subsequent commands which are intended to operate on a message list will be ignored.

d`elete` [*msglist*]
Deletes the specified messages from the current mailbox. If the `mailx` variable `autoprint` is set, the message following the last deleted message is printed.

A deleted message can be restored with `undelete` during a `mailx` session.

di`scard` [*field* ...]
ig`nore` [*field* ...]
Suppresses printing of the specified header fields in the output if they appear at the start of a line and come before a colon, e.g.: `Cc:`, `Date:`, `Status:`, `Subject:`, `To:`. You do not need to include the colon, and `mailx` ignores the case of the letters in *field*.

`discard` has an effect on the `mailx` commands `next`, `pipe` (or |), `print`, `type`, `~f` and `~m`, but on `Print`, `Type` `~F` and `~M`.

No fields are suppressed when messages are saved.

The effect of `discard` can be undone with `undiscard` or `unignore`. `retain` suppresses the effect of `discard`: all fields are ignored except those explicitly specified.

field not specified:

`discard` displays the current list of fields being ignored, if any.

dp [*msglist*]
dt [*msglist*]
(delete and print) Deletes messages from the mailbox and displays the message that follows the last one deleted.

A deleted message can be restored with `undelete` during a `mailx` session.

ec`ho` *string ...*
Echoes *string* on standard output (like the Reliant UNIX `echo` command).

The value of an environment variable can be accessed as `$name`. `echo` always displays the value of

the environment variable, even if a `mailx` variable of the same name has been defined.

e`dit` [*msglist*]

Invokes the editor specified with the `mailx` variable `EDITOR` (default editor: `ed`) and loads the specified messages.

The edited message will be available in the mailbox again at the end of the editing session.

The text is edited in a temporary file named `/tmp/Rz$$`, where `$$` is the process ID of the `mailx` process (see **FILES**).

The `edit` command is not permitted in a command file.

ex`it`
x`it`

Exits from `mailx` without changing the current mailbox, i.e.

- deleted messages are restored,
- read messages are not saved in `$HOME/mbox`, and
- edited messages retain their original status.

Also refer to the `mailx` command `quit`.

fi`le` [*file*]
fold`er` [*file*]

Quits the current mailbox (like `quit`) and reads in the specified file as another mailbox. The appropriate header lines are displayed.

file Name of the mailbox to be processed, or the following metanotation:

`%` the current mailbox

`%`*login_name*
the default mailbox of the named user (`/var/mail/$USER`)

`#` the previous mailbox

`&` the user-specific mailbox (`$HOME/mbox` or the mailbox defined by the variable `$MBOX`)

file not specified:

`mailx` remains in the current mailbox and simply reports the number of messages in it.

`fi %` closes and reopens the current mailbox. This allows you to read any new messages that have arrived during your `mailx` session.

folders

Lists the names of all files in the directory defined by the `mailx` variable `folder` (messages are saved and recorded in this directory by `mailx`).

fo`llowup` [*message*]

Replies to the specified message like the `mailx` command `reply`.

`mailx` switches to send mode and considers the recipients to be

- the author of the specified message, i.e. the entry in the "From" field is taken over to the "To" field,
- the other recipients of the message, i.e. the entries in the "To" field are transferred to the "To" list, while those in the Cc field are copied to the Cc list.

`mailx` takes the contents of the subject field from the corresponding field of *message* and precedes it with `Re:`. If there is no subject entry, you are prompted to enter one.

When you have completed your input, `mailx` sends the message.

In contrast to `reply`, `followup` records the message in a file named after the recipient (with the network path removed). The storage location of this file depends on whether the `mailx` variables `folder` and `outfolder` are set. If both are set, the file is stored in the directory defined by the `folder` variable. Otherwise it is stored in the current directory. The file is extended if it already exists.

The `followup` command is not permitted in a command file.

F`ollowup` [*msglist*]

Replies to the first of the messages specified in the message list in the same way as the `mailx` command `Reply`.

`mailx` switches to send mode and sends the reply to each sender of a message in *msglist*.

`mailx` takes the contents of the subject field from the corresponding field of the first message in *msglist* and precedes it with `Re:`. If there is no subject entry, you are prompted to enter one.

When you have completed your input, `mailx` sends the message.

In contrast to `reply`, `Followup` records the message in a file named after the sender of the first message (with the network path removed). The storage location of this file depends on whether the `mailx` variables `folder` and `outfolder` are set. If both are set, the file is stored in the directory defined by the `folder` variable. Otherwise it is stored in the current directory. The file is extended if it already exists.

The `Followup` command is not permitted in a command file.

f`rom` [*msglist*]

Prints the header summary for all specified messages on the standard output.

g`roup` *alias-name recipient ...*

Declares aliases for the given recipients (see `alias`).

h`eaders` [*message*]

Displays the screen page of headers which includes the specified *message*.

message not specified:

`mailx` displays the first 20 headers, or the number defined by the `screen` variable.

help Displays a summary of the `mailx` commands (see also `?`).

ho`ld` [*msglist*]
pre`serve` [*msglist*]

Holds (i.e. preserves) the specified messages in the default mailbox.

The messages are marked `H` in the header and remain in the mailbox, even if they have been read or saved.

The effect of `hold` can be undone with `touch` (and vice versa).

It is possible to include `hold` and `preserve` commands in a command file, but then any subsequent commands which are intended to operate on a message list will be ignored.

i`f` *mode*
command_list1
el`se`
command_list2
en`dif`

If construct which selects a command list to execute on the basis of the specified mode.

mode *mode* is the mode (send or read) in which you invoked `mailx`. It can be:

- `s` (send) *command_list1* is executed if you invoked `mailx` in send mode; otherwise, *command_list2* is executed.
- `r` (read) *command_list1* is executed if you invoked `mailx` in read mode; otherwise, *command_list2* is executed.

command_list1
command_list2

Lists of `mailx` commands. Commands which are not permitted in command files are also not permitted here, i.e.: `!`, `edit`, `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `shell` and `visual`.

`if`, `else`, `endif` and all the commands in the command lists must each be on a separate line.

ig`nore` [*field* ...]
Suppresses printing of the specified header fields (see `discard`).

l`ist` Lists the names of all available `mailx` commands on standard output (also refer to `help` and `?`).

m`ail` *recipient* ... Sends a message to *recipient*.

`mailx` switches to send mode and mails the message as soon as you have finished entering the text.

If the `mailx` variable `record` is set, the message will be written to the file defined there. If the file already exists, it is extended.

The `mail` command is not permitted in a command file.

M`ail` *recipient* ...
Sends a message to *recipient*.

`mailx` switches to send mode and mails the message as soon as you have finished entering the text.

`mailx` records your message in the current directory in a file named after the recipient. If the file already exists, it is extended.

The `Mail` command is not permitted in a command file.

mb`ox` [*msglist*]
Writes the listed messages to the user-specific mailbox when the current mailbox is closed and then deletes them from the current mailbox, even if they have not been read. All such messages are marked `M` in the header summary.

The user-specific mailbox is `$HOME/mbox` or the file defined by the variable `MBOX`.

n`ext` [*message*]
Goes to the next message containing *message* in the header summary. For *message* you can enter characteristics as for a message list.

This is useful e.g. for jumping to the next message of a specific sender (`next` *login_name*).

message not specified:

`mailx` displays the message following the current one.

`next` otherwise works in the same way as `print`.

pi`pe` [[*msglist*] *shell_command*]
| [[*msglist*] *shell_command*]
Pipes the specified messages to the standard input of the given *shell_command*.

The messages are marked as read (`O`) in the header summary. If the `mailx` variable `page` is set, a form feed character is inserted after each message (FF = CTRL-L = X'0C').

No argument specified:

The default message is the current message, and the default command is the one specified by the `mailx` variable `cmd`. If `cmd` is not set, the `pipe` command is ignored.

pre`serve` [*msglist*]
Preserves the specified messages in the default mailbox (see `hold`).

p`rint` [*msglist*]
t`ype` [*msglist*]
Prints the specified messages on the standard output.

The messages are marked as read (`O`) in the header summary. They are copied to the user-specific mailbox upon termination of the default mailbox and then deleted from the current one. The user-specific mailbox is `$HOME/mbox` or the file defined by the variable `MBOX`.

If the `mailx` variable `crt` is set, messages longer than the number of lines specified by the `crt` variable are paged through the Reliant UNIX command `more`. You can use the variable `PAGER` to specify a Reliant UNIX command other than `more`.

P`rint` [*msglist*]
T`ype` [*msglist*]
Like `print`, except that the whole header is always displayed, i.e. `Print` overrides the effect of `discard` and `ignore`.

q`uit` Exits from `mailx`, closing the currently processed mailbox.

If the default mailbox was being processed, the following applies:

- Read messages (`O`) and messages processed with `mbox` (`M`) are copied to the user-specific mailbox and then deleted. The user-specific mailbox is `$HOME/mbox` or the file defined by the variable `MBOX`.
- Unread messages (`U`) and messages processed with `hold` or `preserve` (`P`) are retained in the default mailbox.
- Explicitly saved messages (`S`) are deleted from the default mailbox.

Also refer to the `mailx` commands `exit` and `xit`.

r`eply` [*message*]
r`espond` [*message*]
Replies to the indicated message.

`mailx` switches to send mode and considers the recipients to be

- the author of the specified message, i.e. the entry in the "From" field is taken over to the "To" field,
- the other recipients of the message, i.e. the entries in the "To" field are transferred to the "To" list, while those in the Cc field are copied to the Cc list.

`mailx` takes the contents of the subject field from the corresponding field of *message* and precedes it with `Re:`. If there is no subject entry, you are prompted to enter one.

When you have completed your input, `mailx` sends the message.

Unlike `followup`, `reply` does not automatically create a file to record your reply. It only does so if the `mailx` variable `record` is set, in which case the message will be written to the file defined there. If the file already exists, it is extended.

The `reply` and `respond` commands are not permitted in a command file.

R`eply` [*msglist*]
R`espond` [*msglist*]
Replies to the first message in the message list.

`mailx` switches to send mode and sends the response to the sender of each message in *msglist*.

`mailx` takes the contents of the subject field from the corresponding field of the first message in *msglist* and precedes it with `Re:`. If there is no subject entry, you are prompted to enter one.

When you have completed your input, `mailx` sends the message.

Unlike `Followup`, `Reply` does not automatically create a file to record your reply. It only does so if the `mailx` variable `record` is set, in which case the message will be written to the file defined there. If the file already exists, it is extended.

The `Reply` and `Respond` commands are not permitted in a command file.

ret`ain` [*field* ...]
displays only the specified fields of the message header. The other fields are suppressed. `retain` displays the specified fields as well, if they are contained in the list of fields to suppress, i.e. `retain` overrides the effects of `discard` or `ignore`.

field not specified:

`retain` displays the current list of fields to display if available.

s`ave` [[*msglist*] *file*]
Saves the indicated messages in the named *file*, extending it if it already exists.

The messages are marked as saved (`S`), which means that they are deleted from the default mailbox as soon as you quit `mailx`, unless you have set the `keepsave` variable).

No argument specified:

`mailx` appends the current message to the end of `$HOME/mbox`.

S`ave` [*msglist*]

Saves the specified messages in a file in the current directory. The file name is derived from the name of the sender of the first message in *msglist* ("From" entry; network addresses are removed). If the file already exists, it is extended.

The messages are marked as saved (`S`), which means that they are deleted from the default mailbox as soon as you quit `mailx`, unless you have set the `keepsave` variable).

s`et` [*name*[=*value*]]

Sets the variable *name*.

name

Name of a `mailx` variable or a freely defined variable.

value

Any string or numeric value. `\n` within *value* is interpreted as a newline character, `\t` as a tab.

value not specified:

name is set to the null string.

No argument specified: Prints all set variables and their values. The values are enclosed in double quotes.

You cannot change the values of environment variables. However, if you define an internal variable of the same name, `mailx` will use its value until you reset it (this does not apply to the `mailx echo` command).

The tilde command `~i` *variable* can be used to insert the value of *variable* in the text of a message.

You can delete variables with `unset` or `set no`*name*.

sh`ell`

Invokes by default the command interpreter specified in the `SHELL` variable. If `SHELL` is not set, `/bin/sh` will be invoked.

You can specify a different command interpreter with the `mailx` variable `SHELL`.

The `shell` command is not permitted in a command file.

si`ze` [*msglist*]

Displays the size of the specified messages on standard output in the form

message_number`:` *number_of_characters*.

so`urce` *file*

Reads the specified file as a command file and executes the `mailx` commands in it. `mailx` then returns to the interactive mode (see **mailx command and startup files** below).

to`p` [*msglist*]

Displays the first 5 lines of the header for each specified message on the standard output. You can change the number of lines displayed with the `mailx` variable `toplines`.

tou`ch` [*msglist*]

Causes the specified messages to be treated as read, i.e. they are copied to the user-specific mailbox upon termination of the default mailbox and then deleted from the current one. The user-specific mailbox is `$HOME/mbox` or the file defined by the `MBOX` variable.

This does not apply to messages that were saved with `save` or `Save`.

`touch` cancels the effect of `hold` and vice versa.

t`ype` [*msglist*]

Prints the specified messages on standard output (see `print`).

Type [*msglist*]
Like print, except that the whole header is always displayed (see Print).

unalias [*alias-name* ...]
deletes the specified alias names.

undelete [*msglist*]
Restores the specified messages provided they were deleted during the current session. The messages are marked as read (O).

If the autoprint variable is set, the last restored message is displayed.

undiscard [*field*]
unignore [*field*]
Deletes the specified header fields from the list of fields being ignored.

field not specified:

Deletes the whole of the list of fields being ignored.

unset [*name* ...]
set [no*name* ...]
Deletes the specified variables.

If you delete a variable with the same name as an environment variable, you can access the value of the corresponding environment variable again.

version
Displays the current version and release number of mailx.

visual [*msglist*]
Invokes the editor specified by the variable VISUAL (default editor: vi) and loads the indicated messages.

The edited message is placed in the mailbox at the end of the editing session.

The text is processed in a temporary file named /tmp/Re$$, where $$ is the process ID of the mailx process.

The visual command is not permitted in a command file.

write [*msglist*] *file*
Writes the specified messages in the named *file*. If the file already exists, it is extended.

write does not copy the header and the trailing blank line.

The messages are marked as saved (S) in the header summary. They are deleted from the default mailbox as soon as you quit mailx, unless you have set the keepsave variable.

xit Exits from mailx without changing the current mailbox (see exit).

z[±] Scrolls the header display one page forward (z+) or back (z–). The number of lines per page is set by the screen variable. If screen is not set, 20 lines are displayed by default.

± not specified:

Same as z+.

Functionality in read mode

mailx checks the contents of the default mailbox /var/mail/$USER, where all incoming messages arrive.

mailx then processes startup files. These files may be used to initialize mailx variables, for example (see **mailx command and startup files** below).

If no mail is present, mailx issues the message:

No mail for *login_name*

If mail has been received, mailx responds with a message line, an overview of all messages currently in the mailbox, and the mailx prompt ?. You can now enter mailx commands. You can use the ?, help and list command to get a list of all the available commands.

Headers

When you call `mailx` or use one of the `mailx` commands `from`, `headers` or `z`, `mailx` displays a header line for each message present. A header can have up to 9 blank-separated fields, e.g.:

```
N   1 marvin    Mon Sep    21 13:05    10/164    Beeblebrox
```

These fields refer to:

`N` Processing status (see next subsection)

`1` Message number. The messages are renumbered each time `mailx` is invoked. The oldest message is numbered 1.

`marvin`
Sender

`Mon Sep 21`
Date message received

`13:05`
Time message received

`10/164`
Size of message in lines/characters

`Beeblebrox`
Title (first 25 characters of subject entry)

Processing status

The processing status is the entry in the first header field. The status will be one of the following:

`O` (old) The message has been read. It will be stored in `$HOME/mbox` when you quit `mailx` or close the default mailbox.

`U` (unread) The message has not yet been read. It will be preserved in the current mailbox when you quit `mailx` using `quit`.

`N` (new) The message has arrived since you last called `mailx` or changed mailboxes.

`M` (mbox) The message has been saved with `mbox`.

`H` (hold) The message has been marked by the `hold` or `preserve` command. It will stay in the default mailbox when you close it.

`S` (save) The message has been saved with a `save`, `Save` or `write` command. It will be deleted from the default mailbox when you close it.

`>`*c* This is the current message. This is the message referenced by `mailx` commands if you do leave *msglist* unspecified. The character *c* stands for any of the above status characters.

User-specific mailbox

Messages are written to the user-specific mailbox if

- you have read them but not deleted or explicitly saved them
- you have manipulated them with `mbox` or `touch`
- you have switched from the default mailbox to another mailbox using the *file* or `folder` command
- you have quit `mailx` with `quit`

except where the variable `hold` is set (then the messages remain in the standard mailbox).

The user-specific mailbox is `$HOME/mbox` or the file defined by the variable `MBOX`. The file is extended if it already exists. If you use the `-f` when you call `mailx`, you can process this file with `mailx` commands in exactly the same way as the default mailbox.

Format 2: Send mode

`mailx` [*option* ...] *recipient* ...

No option specified:
`mailx` behaves as described in **Functionality in send mode**.

`-d` (debug) `mailx` turns on debugging output.

`-F` (file) `mailx` records all outgoing messages in a file named after the first specified recipient. This file is created in your home directory and can be processed with `mailx` like a mailbox.

`-F` not specified:

`mailx` searches for the record file defined in the `mailx` variable `record`. Nothing is recorded if this variable is not set.

`-h` *n* (hop) `mailx` is to make no more than *n* network "hops" from one terminal to another. This is intended to enable network software to avoid infinite delivery loops.

This option is ignored if the delivery program used is `/bin/rmail` (see `mailx` variable `sendmail`).

`-i` `mailx` ignores the `SIGINT` signal (see also the `mailx` variable `ignore`).

`-n` `mailx` does not initialize from the global startup file `/etc/mail/mailx.rc` (see **mailx command and startup files** below).

`-r` *address*

(return) `mailx` passes the specified *address* to the delivery program. All tilde commands are disabled. If the message is undeliverable, it is sent (or returned) to *address*.

This option is ignored if the delivery program used is `/bin/rmail` (see `mailx` variable `sendmail`).

`-s` *subject*

(subject) `mailx` enters *subject* in the `Subject:` header field of the header summary. This allows you to indicate the subject of the message.

subject

Any string. If the string includes blanks or special characters, it must be enclosed in double quotes.

`-U` (UUCP) `mailx` converts UUCP-style addresses to Internet style. The setting of the `conv` environment variable is ignored. Note that the Internet standard is not yet fully supported.

`--` If *recipient* begins with a dash (-), the end of the command-line options must be marked with `--`.

recipient

One or more recipients. *recipient* can be:

- alogin name on the local system or a network address if the computer is connected to a network
- an alias group (see `mailx` read-mode command `alias`)
- apipe symbol followed by a shell command

If *recipient* begins with a pipe symbol (|), the rest of the name is taken to be a shell command to pipe the message through. Thus you could use the `lp` command to send a copy of outgoing mail to a printer.

mailx commands in send mode (tilde commands)

Input format

Apart from being preceded by an escape symbol, `mailx` send-mode commands have the same format as read-mode commands.

[*command*] [*msglist*] [*argument* ...]

`~` Tilde as escape symbol. The `mailx` variable `escape` can be used to define a different character as the escape symbol.

command
msglist
argument

As described above under **mailx commands in read mode, Input format**.

Functional overview

This section provides an overview of all `mailx` send-mode commands, grouped by function. Some

commands may appear more than once.

The overview is followed by descriptions of all the commands in alphabetical order.

Help functions

`~?` Display summary of tilde commands

`~p` Display message being entered

Terminating/aborting text input

`~.` Terminate input and send

`~x` Abort input and do not send

`~q` Abort input and save but do not send

Inserting values of variables, old messages and files

`~a` Insert value of `sign` variable

`~A` Insert value of `Sign` variable

`~i` Insert value of `mailx` or environment variable

`~d` Insert contents of `$HOME/dead.letter`

`~f` Insert old messages

`~F` Insert old messages

`~m` Insert old messages

.TP `~M Insert old messages`

`~r` Insert contents of named file

`~<` Insert contents of named file

Invoking command interpreters, running shell commands

`~!`
Escape to shell

`~<!` Run shell command and insert output in text

`~|` Pass text to shell command and replace with command output

Displaying text

`~p` Display message being entered

Editing text

`~e` Call editor to edit text (default: `ed`)

`~v` Call editor to edit text (default: `vi`)

Modifying mailing lists

`~b` Add names to Bcc list

`~c` Add names to Cc list

`~t` Add names to To list

`~h` Edit To, Subject, Cc and Bcc fields

Manipulating the message header

`~c` Add names to Cc list

`~t` Add names to To list

`~h` Edit To, Subject, Cc and Bcc fields

`~s` Replace contents of Subject field

Recording text

`~w` Write message being entered to file, omitting header

`~q` Abort input and save text but do not send

Send-mode commands which expect mailx to have been called in read mode

`~_` Execute mailx command

`~:` Execute mailx command

`~f` Insert old messages

`~F` Insert old messages

`~m` Insert old messages

`~M` Insert old messages

Executing mailx read-mode commands

`~` Execute mailx read-mode command

`:` Execute mailx read-mode command

Descriptions in alphabetical order

`mailx` commands in send mode (tilde commands) must start with an escape symbol in column one. The default escape symbol is a tilde (~), but this can be redefined with the `mailx` variable `escape`.

Tilde commands are not permitted in a command file.

`!`*shell_command*

Passes the specified *shell_command* to the command interpreter specified in the `SHELL` variable (by default) for execution. If `SHELL` is not set, `/bin/sh` will be invoked.

You can use the `mailx` variable `SHELL` to specify a different command interpreter.

`~.` Terminates message input and sends the message.

If you are working via `rlogin` on a remote computer, this tilde command will be interpreted as an instruction to clear down the connection, with the result that the remote session will be terminated immediately. Here are some possible solutions:

- redefine the escape symbol with the `mailx` variable `escape`
- set the `mailx` variable `dot`, so that you can terminate input simply with a dot in column one
- terminate input with <CTRL-D>

`~:`*mailx-read-command*

`~_`*mailx-read-command*

Executes the specified `mailx` read-mode command.

You must have invoked `mailx` in read mode (and then switched to send mode with a `mailx` command such as `mail`). Otherwise, `mailx` will only execute commands that have nothing to do with the processing of a mailbox (e.g. `set` or `exit`).

The underscore in the second format is mandatory.

`?` Displays a summary of all tilde commands.

`~a` (autograph) Inserts the value of the `mailx` variable `sign` in the message.

`~A` (autograph) Inserts the value of the `mailx` variable `Sign` in the message.

This enables you to define an alternate sign-off string, for example.

`~b` *name ...*

(blind carbon copy) Adds one or more names to the blind carbon copy (Bcc) list. The Bcc list contains the names of additional recipients of the message. These names are not included in the header.

`~c` *name ...*

(carbon copy) Adds one or more names to the carbon copy (Cc) list. The Cc list contains the names of additional recipients. These names are included as part of the header information (the Cc entry).

`~d` (dead letter) Reads the contents of the file `$HOME/dead.letter` into the message. This file contains messages which `mailx` could not send or which you aborted with `q`.

`~e` (e – editor) Invokes the editor specified by the variable `EDITOR` (default editor: `ed`) and loads the partial

message. Input of the partial message may be continued after the editing session.

`~f` [*msglist*]
(file) Inserts the specified messages, without alteration, into the message text.

This command is only executed if `mailx` was invoked in read mode (**Format 1**).

`~F` [*msglist*]
works like `~f`, but inserts always the whole message header. `discard`, `ignore` und `retain` will be ignored.

`~h` (header) Prompts successively for the following information:

`To:` Recipient

`Subject:`
Subject of the message

`Cc:` Carbon copy list containing additional recipients of the message; the names in this list appear in the Cc field of the header

`Bcc:` Blind carbon copy list. Like Cc, except that the names do not appear in the header

The fields are displayed with existing values (if any), which you can edit as if you had just entered them.

`~i` *variable*
Inserts the value of the named *variable* into the message text. The named *variable* can be a `mailx` variable or an environment variable.

`~m` [*msglist*]
(move) Inserts the indicated messages into the text. The contents of the `indentprefix` variable are inserted in front of each line.

This command is executed only if `mailx` was invoked in read mode (**Format 1**).

`~M` [*msglist*]
works like `~f`, but inserts always the whole message header. `discard`, `ignore` und `retain` will be ignored.

`~p` (print) Displays the message being entered.

`~q` Quits input mode by simulating an interrupt. The input text is not aborted but saved in the `$HOME/dead.letter` file. This tilde command has the same effect as the <DEL> key except that it cannot be suppressed with the `ignore` variable.

`~r` *file*
`~r !`*shell_command*
`~<` *file*
`~<!` *shell_command*
(read) Inserts the contents of *file* or the output of *shell_command* into the message text.

`~s` *string ...*
(subject) Sets the Subject field of the header to the specified *string*. Multiple blank-separated strings do *not* need to be quoted.

`~t` *recipient ...*
(to) Adds the indicated names of one or more recipients to the "To" field of the header. Multiple names must be separated by blanks.

`~v` (vi editor) Invokes the screen editor identified by the variable `VISUAL` (default editor: `vi`) and loads the partially entered message. Input of the edited message can be continued on completion of the editing session.

`~w` *file*
(write) Copies the partially entered message, without the header, into the specified file.

`~x` Exits, aborting the message being entered. The partially entered message is neither sent nor saved.

`~|` *shell_command*

Pipes the current text of the message to the standard input of *shell_command*.

If the shell command returns an exit status of 0, the current text is replaced by the output of the command.

By default the command parameter will be passed to the command interpreter specified in the `SHELL` variable. If `SHELL` is not set, `/bin/sh` will be invoked.

Functionality in send mode

When you call `mailx`, it first processes startup files. These files may be used to initialize `mailx` variables, for example (see **mailx command and startup files**).

Then, unless you use the `-s` option to specify a message subject, `mailx` displays:

`Subject:`

and expects you to enter the subject of the message. This line, which may consist of up to 1024 characters, is written by `mailx` into the `Subject:` field of the message header. If the subject is too long, `mailx` will print the message `mail: ERROR signal 10`, and the mail will not be delivered.

`mailx` will now be in send mode, which means that you can enter your message text. All `mailx` tilde commands are permitted during text input. They must start in column 1. Once you have sent off a tilde command with <RETURN>, `mailx` redisplays the whole of the command you have entered in the same line, followed by the string `(continue)` when it has finished executing the command. If you use one of the commands for inserting text into your message text (such as `~a`), `mailx` will not echo the text on the screen. You can view the original text and the inserted text with the `~p` command.

`mailx` stores the input text in a temporary file in the `/tmp` directory.

The command . or the <CTRL-D> key signals the end of input.

Tilde commands in read mode

Some tilde commands only offer their full functionality if you call `mailx` in read mode (**Format 1**) and temporarily switch from there to send mode. The commands in question are: `~_` and `~:` (execute mailx command) and `~f`, `~F`, `~m` and `~M` (insert old messages).

The read commands you can use to switch temporarily to send mode so as to send or reply to a message are `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond` and `Respond`.

mailx command and startup files

Command files

Command files are files that contain `mailx` commands. Each `mailx` command must be entered in a separate line. You can execute command files by using the `source` command during a `mailx` session, or you can use them as startup files (see below).

Tilde commands are not permitted in command files, and nor are the commands `!`, `edit`, `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `shell`, and `visual`.

The `copy`, `Copy`, `hold` and `preserve` commands are permitted, but then any subsequent commands which are intended to operate on a message list will be ignored.

If an error occurs in a command file, `mailx` ignores all subsequent commands in the file.

An error also occurs if a message list refers to a non-existent message (see *Example 2*).

Startup files

Startup files are command files that `mailx` processes every time it is invoked, unless you call it in read mode using the `-e` option.

`mailx` first processes the global startup file `/etc/mail/mail.rc`, followed by the private startup file `$HOME/.mailrc`, provided such files exist.

You can redefine the path name of the private startup file with the `MAILRC` variable.

Variables

`mailx` utilizes environment variables, `mailx` variables and freely defined variables.

All variables can be imported, and during a `mailx` session they can be set and reassigned with the `set` command and deleted with `unset`. `mailx` variables, i.e. all variables consisting of lowercase letters only, can only be set within `mailx` (e.g. in startup files). Contents of shell variables with the same names are not inherited by `mailx` variables.

If you use `set` to reassign an imported variable, the new value applies until you change it again, delete it or end your `mailx` session.

The `mailx` command `echo` always references the original value of an imported variable.

`asksub`, `header` and `save` are enabled by default.

Of the variables which can be assigned values, the following have default values:

- `conv=` (disabled, no conversion)
- `DEAD=$HOME/dead.letter`
- `EDITOR=ed`
- `escape=`
- `MBOX=$HOME/mbox`
- `LISTER=ls`
- `PAGER=more`
- `prompt=?`
- `screen=20`
- `sendmail=/bin/rmail`
- `SHELL=/bin/sh`
- `toplines=5`
- `VISUAL=vi`

The references to *Associated commands* in the following list relate to the `mailx` commands particularly affected by the setting or deletion of the variable in question.

mailx variables

`allnet`

`mailx` treats all network names ending with matching login names as identical. Addresses in the form `...computer_name!computer_name!login_name` are treated as network names.

See also the `metoo` variable.

Default value: The variable is not set.

`append`

Appends messages saved in the user-specific mailbox (`$HOME/mbox` by default) to the end of the file.

Associated commands: `copy`, `Copy`, `file`, `folder`, `mbox`, `next`, `print`, `Print`, `type`, `Type`, `quit`, `touch`

Default value: The variable is not set.

`ask`
`asksub`

Causes `mailx` to prompt for the subject when invoked (also refer to option `-s`).

Associated commands: `~h`, `~s`

Default value: The variable is set.

`askbcc`

`mailx` prompts for the Bcc list after the subject is entered.

Associated commands: `~c`, `~h`

Default value: The variable is not set.

`askcc`
: `mailx` prompts for the Cc list after the subject is entered.

  Associated commands: `~c`, `~h`

  Default value: The variable is not set.

`autoprint`
: Displays the next message after the `delete` command, and the restored message after the `undelete` command.

  Default value: The variable is not set.

`bang` A ! within a *shell_command* in `!`*shell_command* or `~!`*shell_command* will be replaced by the last *shell_command* which was invoked by this way.

  Default value: The variable is not set.

`cmd=`*shell_command*
: Sets the default command used by `pipe` and | to the specified *shell_command* (only used if no command is specified in `pipe`).

  Default value: None.

`conv=`*conversion*
: The only *conversion* available is `internet`. `mailx` then converts UUCP addresses to internet addresses (also refer to option `-U`).

  Associated commands: `alias`, `alternates`, `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~b`, `~c`, `~h`, `~t`

  Default value: no conversion

`crt=`*number*
: Pipes message output having more than *number* lines through the command specified by the `PAGER` variable (`PAGER=more` by default).

  Associated commands: `dp`, `dt`, `next`, `print`, `Print`, `type`, `Type`, `~p`

  Default value: The variable is not set.

`debug`
: Enables diagnostics for debugging. If you set this variable, mail will not be delivered.

  Default value: The variable is not set.

`dot` Causes a line consisting solely of a dot in column one to terminate input (instead of the command `~.`).

  Default value: The variable is not set.

`escape=`*c*
: Substitutes *c* for the tilde escape symbol.

  Default value: `~`

`flipr`
: The effects of the commands `reply`, `respond`, `Reply` and `Respond` are exchanged.

  Associated commands: `reply`, `respond`, `Reply`, `Respond`.

  Default value: The variable is not set.

`folder=`*directory*
: If both `folder` and `outfolder` are set, reply texts with the `followup` and `Followup` commands will be recorded in *directory*, not in the current directory. If *directory* does not begin with a slash, `mailx` sets the directory name to `$HOME/`*directory*.

  You can also use the form `+`*filename* to reference these record files in any `mailx` command which accepts file names. `mailx` will then expand the name by prepending *directory*.

  Default value: The variable is not set.

`header`
: Causes the header summary, the current `mailx` version string and the number of messages to be

displayed when `mailx` is invoked.

Default value: The variable is set.

`hold` Preserves read messages in the default mailbox instead of putting them in the user-specific mailbox file (also refer to `MBOX` variable).

Associated commands: `copy`, `hold`, `mbox`, `next`, `preserve`, `print`, `Print`, `quit`, `touch`, `type`, `Type`

Default value: The variable is not set.

`ignore`

Ignores the `SIGINT` signal during message input.

Associated commands: `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~q` (<DEL>)

Default value: The variable is not set.

`ignoreeof`

The end-of-file signal (EOF, <CTRL-D> key) is to be ignored during message input (also refer to the `dot` variable).

Associated commands: `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~.`

Default value: The variable is not set.

`indentprefix=`*string*

During the insertion of a message into a text each line of the message begins with *string*.

Associated commands: `~m`, `~M`

Default value: Tab character.

`keep` The default mailbox should not be deleted when it is empty.

Default value: The variable is not set.

`keepsave`

Messages that are marked as saved (`S`) in the header are not to be deleted from the default mailbox.

Associated commands: `save`, `Save`, `write`

Default value: The variable is not set.

`metoo`

If your own login name appears in the list of recipients ("To" list), it is not deleted from the list. See also the `mailx` command `alias` and the `allnet` variable.

Associated commands: `alias`, `alternates`, `followup`, `group`, `reply`, `respond`, `~h`

Default value: The variable is not set.

`onehop`

When you reply to a message which has been sent to multiple recipients, `mailx` abandons its usual practice of constructing all recipients' addresses relative to the originating author's machine. This improves efficiency when all network sites have direct access to one another.

Associated commands: `followup`, `Followup`, `reply`, `Reply`, `respond`, `Respond`

Default value: The variable is not set.

`outfolder`

If both `outfolder` and `folder` are set, reply files with the `followup` and `Followup` commands will be stored in the directory defined by `folder` (also refer to `record` variable).

If only `outfolder` (or only `folder`) is set, reply files will be stored in the current directory.

Default value: The variable is not set.

`page` Causes a form feed character (FF = CTRL-L = X'0C') to be inserted after each message routed through a `pipe` or `|` command.

Default value: The variable is not set.

`prompt=`*string*

Sets the input prompt for `mailx` read-mode commands to *string*.

Default value: `?`

`quiet`

Suppresses the display of the opening message and version identifier when `mailx` is invoked.

Default value: The variable is not set.

`record=`*file*

Records all outgoing mail in the named *file*. The file is extended if it already exists.

Associated commands: `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~.`

Default value: The variable is not set.

`save` Saves messages that could not be sent, e.g. due to an error or an interrupt from <DEL> during input. These messages are saved in the file specified by the `DEAD` variable.

Associated commands: `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~d`, `~q` (<DEL>)

Default value: The variable is set.

`screen=`*number*

`mailx` displays *number* header lines.

Associated commands: `header`, `z+`, `z-`

Default value: Dependent on the terminal type specified by `TERM`, usually 20.

`sendmail=`*shell_command*

`mailx` uses *shell_command* to send messages.

Associated commands: `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~.`

Default value: `/bin/rmail`

`sendwait`

After a read command, waits for the background mailer to finish before returning control to the user.

Associated commands: `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply, respond`, `Respond`

Default value: The variable is not set.

`showto`

If you send a message and include yourself among the recipients, the name of the first recipient in the recipient list ("To" list) will appear in the header list displayed, not your login name.

Associated commands: `from`, `headers`, `z+`, `z-`

Default value: The variable is not set.

`sign=`*string*

Defines an autograph string to be inserted into the text of a message.

Associated commands: `~a`, `~i`

Default value: The variable is not set.

`Sign=`*string*

Defines an (additional) autograph string to be inserted into the text of a message.

Associated commands: `~A`, `~i`

Default value: The variable is not set.

`toplines=`*number*

`mailx` displays *number* lines from the message header.

Associated commands: `top`

Default value: 5

Environment variables

`DEAD=`*file*

The named *file* is used by `mailx` to save messages that could not be sent, e.g. due to an error or an

interrupt during input. This file is extended if it already exists.

Associated commands: `followup`, `Followup`, `mail`, `Mail`, `reply`, `Reply`, `respond`, `Respond`, `~d`, `~q` (<DEL>)

Default value: `$HOME/dead.letter`

`EDITOR=`*shell_command*
In a `mailx` session, message texts can be edited with the editor named as *shell_command* (also refer to variable `VISUAL`).

Default value: `ed`

`HOME=`*directory*
Home directory.

Specifies the default directory in which `mailx` creates or searches for the files `dead.letter`, `mbox`, `.mailrc`, and the files used to record outgoing messages (see the commands `followup`, `Followup` and `~f`, `~F`).

`LISTER=`*shell_command*
Uses *shell_command* to list files in the `folder` directory.

Associated commands: `folders`

Default value: `ls`

`MBOX=`*file*
file designates the user-specific mailbox in which `mailx` is to save read messages before removing them. Each additional message extends the file (also refer to the `hold` variable).

Associated commands: `copy`, `hold`, `mbox`, `next`, `preserve`, `print`, `Print`, `quit`, `save`, `touch`, `type`, `Type`

Default value: `$HOME/mbox`

`MAILRC=`*file*
file defines the name of the private startup file (default: `$HOME/.mailrc`; see **mailx command and startup files**).

`PAGER=`*shell_command*
Uses the specified *shell_command* for the paging of output that exceeds the number of lines defined in the `crt` variable.

Associated commands: `dp`, `dt`, `next`, `print`, `Print`, `type`, `Type`, `~p`

Default value: `more`

`SHELL=`*shell_command*
Defines the command interpreter used by `mailx` to execute Reliant UNIX commands.

Associated commands: `!`, `shell`

Default value: `/bin/sh`

`TERM` Contains information about the terminal type. The `TERM` variable will be evaluated if the `mailx` variable `screen` is not set.

`USER` The `USER` variable tells `mailx` the login name of the user, to enable it to identify the default mailbox etc.

`VISUAL=`*shell_command*
In a `mailx` session, message texts can be edited with the editor named as *shell_command*.

Associated commands: `visual`, `~v`

Default value: `vi`

NOTE ON ERROR MESSAGES

`mailx`'s error messages are largely self-explanatory.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

LC_TYPE governs character classes and character conversion (shifting).

LC_COLLATE governs the collating sequence.

The LC_TIME environment variable governs the format of date and time strings.

If LC_MESSAGES, LC_CTYPE, LC_COLLATE or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Replying to a message

User "marvin" receives the message you have mail, so he calls mailx without options. He sees a few lines of text and some header lines, has all his messages displayed one after the other (using <RETURN>, for example) and finally replies to the fourth message. The text of his reply is to be recorded in the current directory (followup or fo). While in send mode, he changes the Subject entry assigned automatically by mailx (~s) and inserts his sign-off string, as defined in his sign variable, at the end of the text (~a). Before sending his letter (~.), he has it redisplayed (~p) and then quits mailx with xit or x so as to retain all his messages in the mailbox.

```
mailx
mailx version 4.0 Type ? for help
"var/mail/marvin": 4 messages 4 new
>N 1   arthur     Fri Sep  6   9:21    13/373      betelgeuse
 N 2   frodo      Fri Sep  6  12:00    13/365      sysadm
 N 3   peter      Mon Sep 16  10:01     9/232      qed
 N 4   benny      Tue Sep 17  16:43    21/593      Project S
 <RETURN> .
 .
 fo benny
 To: <benny>
 Subject: Re: Project S
 ~s Final report on P S~s Final report on P S
 .
 .
 ~a~a
 ~p~p
 --------
 Message contains:
 To: <benny>
 Subject: Final report on P S
 Hi there, Ben!
 Thanks for the minutes. What I really need is the final report
 itself, and not just Real Soon Now, but tomorrow at the latest.
 I need a printed version, not just a file.
 Regards,
 (-: marvin :-)
 (continue)
 ~.
 ? x
 Held 4 messages in /var/mail/marvin
 $
```

Example 2

Example of a startup file

The following startup file sets variables and prints all messages from `winnie` (with `lpr`) when `mailx` is invoked in read mode.

```
# Processing variables
set page crt=24 cmd=lpr VISUAL
set sign="\n\tFord Prefect\n\tSales Division\n\tDetroit"
# Sender: Network system administrator
alias sys root@orlando root@annapolis root@chicago
# Prints specific mail
if r
pipe winnie lpr
from winnie
endif
```

Note that `mailx` terminates the script if any command cannot be executed. This might be the case here with the `pipe` command if no messages from `winnie` were present. In other words, `mailx` then would not execute the `from` command (or any others that might follow).

FILES

`/etc/mail/mail.rc`
: Global startup file

`$HOME/.mailrc`
: Private startup file

`/var/mail/$USER`
: The default mailbox that is searched by `mailx` for incoming messages.

`$HOME/mbox`
: User-specific mailbox in which `mailx` saves messages that have been read.

`$HOME/dead.letter`
: File that is used by `mailx` to save messages that could not be sent, e.g. due to an error or an interrupt with the <DEL> key during input. If the file already exists, it is overwritten.

./login_name
: Files created in the current directory with names derived from a login name. These files are created by `mailx` in response to the following commands:

  `Copy`, `followup`, `Followup`, `Save`.

  You can also select some other directory instead of the current directory (see the `mailx` variables `folder` and `outfolder`).

`/tmp/R[emsxz]*`
: Temporary files.

`/tmp/Rz$$`
: Temporary file used by the commands `edit`, `visual`, `e`, and `v`. `$$` is the process ID of the `mailx` process.

`/usr/share/lib/mailx/mailx.help*`
: Help message files.

SEE ALSO

`ed(1)`, `ls(1)`, `mail(1)`, `more(1)`, `sh(1)`.

User's Reference Manual

makekey(1)

NAME

`makekey -` generate encryption key

SYNOPSIS

`/usr/lib/makekey`

DESCRIPTION

`makekey` is an encryption routine that is primarily used to encrypt passwords [see `passwd(4)`].

The `makekey` command reads from standard input and is normally used in pipelines.

Functionality

`makekey` processes 10 bytes of input. The last two input bytes (the `salt`) are used to select the encryption algorithm with which the first eight bytes (the `input key`) are encrypted.

The first eight bytes can be arbitrary ASCII characters. The last two are best chosen from the set of digits, letters, dot (.) and slash (/).

The salt characters are repeated as the first two characters of the output. This is followed by 11 bytes representing the encrypted form.

Caution:

`makekey` can produce different results depending upon whether the input is typed at the terminal or redirected from a file.

EXAMPLES

The following shell script named `pass.scr` defines a password using `/usr/lib/makekey` and then prompts the user to enter this password.

```
:
SECRET_KEY='A/'
#
#          READ AND ENCRYPT A NEW PASSWORD
#
size=0
while [ "$size" -lt "8" ]
do echo -n 'Please enter a password (at least 8 characters):'
   stty -echo
   read password
   echo
   stty echo
   set `echo -n $password | wc -c`
   size=$1
done
#
# Strip password to max. 8 characters:
#
password="`echo $password | awk '{x=substr($0,1,8); printf("%8s\n",x);}'`"
#
# Append secret key to password and encrypt password
#
password="`echo $password$SECRET_KEY | /usr/lib/makekey`"
#
# Strip secret key from encrypted password
#
CODED_PWD="`echo $password | awk '{x=substr($0,3,11);printf("%11s\n",x);}'`"
```

```
#
#            USING AN ENCRYPTED PASSWORD
#
echo -n 'Enter the password:'
stty -echo
read password
echo
stty echo
#
# Strip password to max. 8 characters:
#
password="`echo $password | awk '{x=substr($0,1,8);printf("%8s\n",x);}'`"
#
# Append secret key to password and encrypt password
#
password="`echo $password$SECRET_KEY | /usr/lib/makekey`"
if [ "$password" != "$SECRET_KEY$CODED_PWD" ]
then echo 'Invalid password'
else echo 'The entered password was correct !!!'
fi
```

SEE ALSO

crypt(1), ed(1), vi(1), passwd(4).

User's Reference Manual

man(1)

NAME

`man, apropos, whatis, findman -` display on-linemanual pages

SYNOPSIS

`man` [`–e`] [*section* | `–s` *section*] [`––`] *title ...*

`man –r` [`––`] *man_src_file ...*

`apropos` [`–e`] [`––`] *regexp_keyword ...*

`man –k` [`–e`] [`––`] *regexp_keyword ...*

`whatis` [`––`] *regexp_keyword ...*

`findman` [`–e`] [`––`] *regexp_keyword ...*

`man –f` [`–e`] [`––`] *regexp_keyword ...*

DESCRIPTION

The `man` command displays reference manual pages that document commands, system calls, subroutines, file formats, etc., in the base product and optional products.

A *title* can exist in more than one *section*. When a *section* is specified, `man` searches for the specified *title* only in that *section*. The *title* and *section* searches are made without regard to case. The *section* is specified by a number or number/letter combination. Sections are typically associated with general topics, as described below:

`1` = Commands/Applications

`1M` = System Administration Commands

`2` = System Calls

`3` = Subroutines

`4` = File Formats

`5` = Miscellaneous Facilities

`7` = Special Files

`8` = System Maintenance

If the *section* is not specified, `man` searches all sections (first commands, then functions) and prints all manual pages found. If the *section* does not begin with a digit, `–s` must be used. If no manual pages are found, `man` prints an error message.

If the preformatted version of interest is out of date or missing, `man` reformats it prior to display. If directories for the preformatted versions are not provided or creatable, `man` reformats a page whenever it is requested; it uses a temporary file to store the formatted text during display.

The commands `apropos`, `whatis` and `findman` are short forms for the `man` command with specific options (see below).

OPTIONS

`–e` Forces the *section* match to be exact (default for `whatis`). If `–e` is specified, only the man pages in the specified *section* will be displayed. If the *section* does not begin with a digit, `–s` must be used.

`–f` Displays one-line summaries whose *title keyword* to the left of the separator "–" matches *regexp_keyword* (default for `findman` and `whatis`). See *Example 1*.

`–k` Displays one-line summaries whose *title and/or abstract* to the left and right of the separator "–" match

regexp_keyword (default for `apropos`). See *Example 2*.

`-r` Arguments are taken to be files containing `man` macros. These files are formatted and written to standard output. This option also ignores the `mandb` database and existing `cat` pages.

`-s` Specification of a section. This option must be used to search a section which does not begin with a number.

`--` If *title* or *regexp_keyword* begin with a dash (-), the end of the command-line options must be marked with --.

man_src_file
File containing `man(5)` macros.

regexp_keyword
Extended regular expression [see `expressions(5)`] to select a title or abstract.

Warning: A regular expression should be enclosed in single quotes (') for the purposes of security.

section
Section to be searched. See *Example 3*.

title old text: Name of an entry in the reference manual (e.g. a command, a system call, etc). This name can be specified as a simple regular expression [see `expressions(5)`].

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Search for manual page titles (to the left of the separator "-" only) containing the regular expression "mount":

```
findman mount

mount (1M-nfs)  - mounting remote NFS resources
mount (1M-hs)   - mount hs file systems
mountd (1M)     - NFS mount request server
mount (1M)      - mount or unmount file systems and remote resources
mount (1M-ufs)  - mount ufs file systems
mount (1M-vxfs) - mount a Veritas File System (VxFS)
mountall (1M)   - mount, unmount multiple file systems
mountfsys (1M)  - mount, unmount a file system
mount (2)       - mount a file system
```

Example 2

Search for manual page titles containing the regular expression "mount". This search also covers the short explanation, i.e. to the left *and* right of the separator "-".

```
apropos mount

automount (1M)  - automatically mount NFS file systems
dfmounts (1M)   - display information on mounted NFS resources
mount (1M-nfs)  - mounting remote NFS resources
mount (1M-hs)   - mount hs file systems
mountd (1M)     - NFS mount request server
showmount (1M)  - show information on available or mounted NFS resources
unshare (1M)    - make local NFS resource unavailable for mounting by
remote systems
```

```
mnttab (4)       - mounted file system table
mount (1M)       - mount or unmount file systems and remote resources
mount (1M-ufs)   - mount ufs file systems
mount (1M-vxfs)  - mount a Veritas File System (VxFS)
mountall (1M)    - mount, unmount multiple file systems
mountfsys (1M)   - mount, unmount a file system
setmnt (1M)      - establish mount table
umount (1M)      - mount or unmount file systems and remote resources
umountall
(1M)   - mount, unmount multiple file systems
umountfsys (1M)  - mount, unmount a file system
mount (2)        - mount a file system
umount (2)       - unmount a file system
```

Example 3

Displays *only* the Veritas version of the command description for "mount":

```
man 1M-vxfs mount
```

Example 4

Displays *all* command descriptions for "mount":

```
man mount
```

NOTES

Some dumb terminals cannot correctly process the vertical motions produced by `eqn(1M)` preprocessing flag. Information indicated by font changes and other phototypsetting directives may be lost.

These commands may take a `–v` which turns on a debug mode. This option, and the output that is produced, is subject to change or removal without notice.

The UCB compatibility utilities:

```
/usr/ucb/man
/usr/ucb/whatis
/usr/ucb/catman
/usr/ucb/apropos
```

are not compatible with the utilities listed in this manual page.

The `manprint` command allows you to print a manual page on a line printer, while maintaining page integrity with the screen output. Further details can be found under `manprint(1)`.

ENVIRONMENT

`LANG` Environment variable for the language setting. Default: `En_US.ASCII` [cf. `locale(1)`, `–a` option].

`MANPATH`
: If set, its value overrides the default search path, `/usr/share/man/mrd/%L:/usr/share/man/mrd:/usr/share/man`. A `%L` in `$MANPATH` will be replaced by the value of `$LANG` when invoking the command. If `$LANG` is not set or not valid, it is replaced by `En_US.ASCII`.

`PAGER` A program to use for interactively delivering `man` output to the screen. If not set, `more –cdEilqws` [see `more(1)`] is used.

FILES

`/usr/share/man/mrd`
: Main directory for the on-line `man` page system.

`/usr/share/man/mrd/$LANG/catman/*`
: Directory for compressed and formatted manual pages (`.Z` files), displayed when you call the `man` command.

`/usr/share/man/mrd/$LANG/.mandb/*`

Directory that serves as a basis for the `mandb` database (`/usr/share/man/mrd/*/mandb`).

If the date of this directory is more recent than that of the `mandb` database file, this file is regenerated and an appropriate message is issued [cf. `catman(1M)`].

`/usr/share/man/mrd/$LANG/mandb`

Database file with the titles of the manual pages, section numbers, file names and short descriptions created on the basis of the `.mandb` directory.

SEE ALSO

`manprint(1)`, `more(1)`, `catman(1M)`, `neqn(1M)`, `nroff(1M)`, `expressions(5)`, `man(5)`.

User's Reference Manual

man(1-ucb)
BSD Compatibility Package

NAME

`man` - display reference manual pages; find reference pages bykeyword

SYNOPSIS

`/usr/ucb/man` [`–`] [`–t`] [`–M` *path*] [`–T`*macro-package*] [[*section*] *title* ...] *title* ...

`/usr/ucb/man` [`–M` *path*] `–k` *keyword* ...

`/usr/ucb/man` [`–M` *path*] `–f` *filename* ...

DESCRIPTION

The `man` command displays information from the reference manuals. It can display complete manual pages that you select by *title*, or one-line summaries selected either by *keyword* (`–k`), or by the name of an associated file (`–f`).

A *section*, when given, applies to the *titles* that follow it on the command line (up to the next *section*, if any). `man` looks in the indicated section of the manual for those *title*s. *section* is a digit (perhaps followed by a single letter indicating the type of manual page). If *section* is omitted, `man` searches all reference sections (giving preference to commands over functions) and prints the first manual page it finds. If no manual page is located, `man` prints an error message.

The reference page sources are typically located in the `/usr/share/man/man?` directories. Since these directories are optionally installed, they may not reside on your host; you may have to mount `/usr/share/man` from a host on which they do reside. If there are preformatted, up-to-date versions in `/usr/share/man/cat?` or `/usr/share/man/fmt?` directories, `man` simply displays or prints those versions. If the preformatted version of interest is out of date or missing, `man` reformats it prior to display. If directories for the preformatted versions are not provided, `man` reformats a page whenever it is requested; it uses a temporary file to store the formatted text during display.

If the standard output is not a terminal, or if the – flag is given, `man` pipes its output through `cat`. Otherwise, `man` pipes its output through `more` to handle paging and underlining on the screen.

OPTIONS

`–t` `man` arranges for the specified manual pages to be `troff`ed to a suitable raster output device [see `troff`(1M)]. If both the – and `–t` flags are given, `man` updates the `troff`ed versions of each named *title* (if necessary), but does not display them.

`–M` *path*
Change the search path for manual pages. *path* is a colon-separated list of directories that contain manual page directory subtrees. When used with the `–k` or `–f` options, the `–M` option must appear first. Each directory in the *path* is assumed to contain subdirectories of the form `man`[1-8] (see also environment variable `MANPATH` below).

`–T` *macro-package*
`man` uses *macro-package* rather than the standard `–man` macros defined in `/usr/lib/tmac/tmac.an` for formatting manual pages.

`–k` *keyword* ...
`man` prints out one-line summaries from the `whatis` database (table of contents) that contain any of the given *keyword*s.

`–f` *filename* ...
`man` attempts to locate manual pages related to any of the given *filename*s. It strips the leading pathname components from each *filename*, and then prints one-line summaries containing the resulting basename or names.

MANUAL PAGES

Manual pages are `troff` or `nroff` source files prepared with the `–man` macro package.

When formatting a manual page, `man` examines the first line to determine whether it requires special processing.

Preprocessing Manual Pages

If the first line is a string of the form:

`'\" X`

where *X* is separated from the `"` by a single SPACE and consists of any combination of characters in the following list, `man` pipes its input to `troff` or `nroff` through the corresponding preprocessors.

`e` `eqn`, or `neqn` for `nroff`

`r` `refer`

`t` `tbl`, and `col` for `nroff`

If `eqn` or `neqn` is invoked, it will automatically read the file `/usr/ucblib/pub/eqnchar` [see eqnchar(5)].

ENVIRONMENT

`MANPATH`
If set, its value overrides `/usr/share/man` as the default search path. The `–M` flag, in turn, overrides this value.

`PAGER`
A program to use for interactively delivering `man`'s output to the screen. If not set, "`more –s`" [see more(1)] is used.

`TCAT` The name of the program to use to display `troff`ed manual pages. If not set, "`lp –Ttroff`" [see lp(1)] is used.

`TROFF`
The name of the formatter to use when the `–t` flag is given. If not set, `troff` is used.

NOTES

The manual is supposed to be reproducible either on a phototypesetter or on an ASCII terminal. However, on a terminal some information (all `troff` requests, for instance) is necessarily lost.

Some dumb terminals cannot process the vertical motions produced by the `e` [eqn(1M)] preprocessing flag. To prevent garbled output on these terminals, when you use `e` also use `t`, to invoke col(1) implicitly. This workaround has the disadvantage of eliminating superscripts and subscripts – even on those terminals that can display them. CTRL-Q will clear a terminal that gets confused by eqn(1M) output.

CAVEAT

The `/usr/ucb/man` command will not display Reliant UNIX man pages. It is included for users who have BSD man pages.

FILES

`/usr/share/man`
Main directory for the on-line `man` page system

`/usr/share/man/man?/*`
Unformatted manual entries

`/usr/share/man/cat?/*`
`nroff`ed manual entries

`/usr/share/man/fmt?/*`
`troff`ed manual entries

`/usr/share/man/whatis`

Table of contents and keyword database

`/usr/ucblib/makewhatis`
Command script to make `whatis(1)` database

`/usr/lib/tmac/tmac.an`
Standard `–man` macro package

`/usr/ucblib/pub/eqnchar`
Special character definitions for `eqn` and `neqn`

SEE ALSO

`apropos(1)`, `cat(1)`, `col(1)`, `lp(1)`, `more(1)`, `refer(1)`, `whatis(1)`, `catman(1M-ucb)`, `eqn(1M)`, `neqn(1M)`, `nroff(1M)`, `tbl(1M)`, `troff(1M)`, `eqnchar(5)`.

User's Reference Manual

manprint(1)

NAME

`manprint -` print reference pages on a line printer maintaining pageintegrity

SYNOPSIS

`manprint` [*section*] *title*

DESCRIPTION

When working with reference pages, or manpages as they are commonly called, users from time to time also need to be able to print out a manpage displayed online with the `man(1)` command to ease legibility. The `manprint` command can be used for this purpose.

`manprint` sends reference pages to standard output in formatted form. In this respect, it works in the same way as the `man` command. In contrast, however, you can easily forward the output of the `manprint` command to a line printer by means of a pipe and in this way print a manpage exactly as it was displayed on the screen (e.g. the manpage `diff3(1)`: `manprint diff3 | lp -d`*lineprinter*). It does not make sense to do this with the output of the `man` command, because the form feeds are different with the print output and the screen output and the resulting print-out does not look very good.

As with the `man` command, you can specify section identifiers or alias names as arguments with the `manprint` command. You should note the following here:

Section identifiers

A manpage can be invoked explicitly with section identifiers with both the `manprint` command and the `man` command. This can be useful if a manpage exists in several *sections* under the same *title* and only differs in terms of the section identifier. The section identifier can always be found in round brackets after the title of a manpage or also at the right and left margin in each header line on the screen. "iconv" is an example of a manpage with several section identifiers, which exists in three variants: `iconv(1)` [user command], `iconv(3C)` [C function] and `iconv(5)` [header file]. If you invoke the manpage with `manprint iconv`, all three descriptions are displayed in succession on the screen (or printed if this has been specified). If, on the other hand, you are only interested in one of these descriptions – e.g. the one for the C function –, you have to specify the section identifier before the manpage title: `manprint 3C iconv`.

The same applies where there are different versions of manpages, for example AT&T and Berkeley (UCB). You can also invoke the required description explicitly in this case. Take, for example, the "chown" command which exists in two variants, i.e. AT&T [`chown(1)`] and UCB [`chown(1-ucb)`]. If, for example, you are only interested in the UCB description, you have to invoke the manpage as follows: `manprint 1-ucb chown`.

See also under *Example 2* and the section **NOTES**.

Alias names

An alias name occurs if the first entry (= *title*) in the section **NAME** is followed by further titles separated by commas.

You can see examples of such alias names in the `string(3C)` manpage, where the primary collective title "`string:`" is followed by fifteen other titles of routines (from `strcat` to `strstr`). Another typical example is the `enable(1)` manpage which also bears the alias name `disable`.

These alias names are practical from the point of view that a manpage can be invoked by means of its alias name as well as by its *title*. The user therefore does not need to know the title under which a description can be found but can use the alias that is know to him or her. As a result, the same `string(3C)` manpage containing the required description is output when you invoke `manprint string` and `manprint strstr`.

See also under *Example 3*.

ARGUMENTS

section

A specific section identifier can be specified optionally for a manpage, e.g. `2`, `1M`, `3C`, `1-ucb`, `1M-vxfs` etc.

If a section identifier is not specified and if several manpages exist with the same title but different section identifiers, all manpages with this title are output.

title

Mandatory specification of a title or alias name for a manpage, e.g. `ksh`, `rksh`, `ctype`, `isalpha` etc.

Wildcards should not be used when specifying the title (`*`, `?`).

Note: It is *not* possible to have two manpages with the same name that can also have the same section identifier (because they are printed in different manuals).

EXAMPLES

Example 1: Redirect output to file and print

```
manprint uniq | tee uniq.man | lp -dlineprinter
```

The manpage for the `uniq(1)` command is redirected with `tee(1)` to the `uniq.man` file and printed at the same time on a line printer.

Example 2: Specify section identifier

There are six different manpages for the `mount` interface: there are descriptions of the `mount` command for different file systems (`hs`, `ufs`, `vxfs` etc.) and a description of the `mount(2)` system call. Since you are only interested in the `mount` command for the Veritas file system, enter the `1M-vxfs` section identifier explicitly as the first argument:

```
manprint 1M-vxfs mount | lp -dlineprinter
```

Only the specific Veritas manpage `mount(1M-vxfs)` will be printed. If you omitted the `1M-vxfs` specification, all 6 `mount` manpages would have been printed all at once.

Example 3: Specify alias name

You want to output the description of the logarithm routine `log1p` but do not know in which manpage it is described. Nevertheless, you enter:

```
manprint log1p | lp -dlineprinter
```

The `exp(3M)` manpage is printed, which also lists and describes `log1p` under several alias names (`expf`, `expm1`, `cbrt` to `sqrtf`).

NOTES

Font settings (italics and courier) in the manpage text are retained in the output and are reproduced by the line printer if the printer supports this, i.e. they are generally printed with underscoring. It is not recommended, however, to redirect the output to a file and then to remove the font settings. This could be done easily with the `col –b` command. The `–b` option of the `col` command removes all escape sequences [see `col(1)`]. As a result, all of the escape sequences are also removed for the form feed and the print output then no longer maintains page integrity with the screen output of the `man` command.

There is one restriction to be noted when specifying a section identifier (*section* argument). In the following case, neither `man` nor `manprint` works properly:

```
manprint 1 echo
```

The `echo` manpage with the section identifier "1" [i.e. `echo(1)`] is not the only manpage output – as was required – rather the manpages `echo(1-ucb)` and `echo(1F)` are also printed. This means that the section identifier "1" is expanded inadmissibly. On the other hand, however, by specifying:

```
manprint 1-ucb echo
```

only the required `echo(1-ucb)` manpage is output.

SEE ALSO

`man(1)` .

User's Reference Manual

md5(1)

NAME

`md5` - calculate a checksum in the form of a fake-proof fingerprint

SYNOPSIS

`md5` *file ...*

`md5` *options* [*file ...*]

DESCRIPTION

Checksums for files can be calculated using the `sum` or `cksum` command. However, both commands are only suitable for detecting transmission errors. The underlying polynomial is extremely simple and in particular allows files to be modified in such a way that the checksum calculated is identical with a previously specified checksum (even if the files are different).

`md5` takes as input a specified string or message or file of arbitrary length and creates a checksum in the form of a 128-bit fingerprint (also termed message digest) as output. To date it has not been possible to create two strings with the same fingerprint or one string with a prespecified fingerprint. This makes `md5` a very safe procedure for producing unique digital signatures.

OPTIONS

The following four options may be used in any combination, except that *file* must be the last object on the command line.

If no option or file is specified, `md5` reads from standard input.

`–s"`*string*`"`
: Outputs the checksum of the specified *string*.

`–t`
: Runs a built-in time trial.

`–x`
: Starts a built-in test suite for comparison with the checksum supplied by the current computer.

file
: Calculates the checksum(s) for the specified file(s).

EXAMPLES

Example 1:

Use the following command to create the `md5` checksum for the string "Siemens Nixdorf":

Input:

```
md5 –s"Siemens Nixdorf"
```

Output:

```
MD5 ("Siemens Nixdorf") = b89058eb500949a2d16f69baf78cf45d
```

Example 2:

A typical sample application is to use the checksum of a software package. If you want to check the `md5` checksum for a software package, apply the command to the relevant package and compare the result with the value from the package documentation.

Input:

```
md5 SIY3A232+.pkg
```

Output:

```
MD5 (SIY3A232+.pkg) = 51f0ac1b6d777b6278facde79a901385
```

SEE ALSO

`cksum`(1), `sum`(1).

User's Reference Manual

mesg(1)

NAME

mesg - permit or deny messages

SYNOPSIS

mesg [y|n]

DESCRIPTION

mesg allows you to control whether other users can send messages to your terminal. You can use mesg either to simply check the status of your terminal or to grant or deny other users the permission to use write to write messages to your terminal.

OPTIONS

No argument specified:

mesg reports the current setting for your terminal and returns an exit status of 0 if messages may be sent to it, or 1 if they cannot.

y Grants other users permission to write messages to your terminal.

This corresponds to the old argument –y which is still supported.

n Denies other users permission to write messages to your terminal. Messages sent via wall or write by the system administrator override this denial.

This corresponds to the old argument –n which is still supported.

EXIT STATUS

0 Messages are receivable

1 Messages are not receivable

>1 Error

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

User heidi, who is logged in at terminal tty001, checks the current setting for this terminal:

```
$ mesg
is n
```

Messages are not receivable at terminal tty001. Thus no messages can be sent to heidi while she is working at terminal tty001:

```
$ write heidi
Permission denied
```

Example 2

heidi can change the setting at terminal tty01 by entering:

```
$ mesg y
```

FILES

`/dev/tty*`
`/dev/term/tty*`
files containing terminal-specific information

SEE ALSO

`talk`(1), `tty`(1), `write`(1), `wall`(1M).

User's Reference Manual

mkdir(1)

NAME

mkdir - make a directory

SYNOPSIS

mkdir [*option* ...] *directory* ...

DESCRIPTION

mkdir is used to create new directories.

mkdir automatically makes the following standard entries in the new directory:

. (dot)
: for the directory itself

.. (dot dot)
: for the parent directory

To be able to use mkdir you have to have write permission in the parent directory.

OPTIONS

No option specified:
: mkdir creates the named directories in mode 777 [see chmod(1)], modified by the current file-creation mode mask [see umask(1)].

–m *mode*
: The access permissions specified in *mode* are used for the new *directory* [see chmod(1)].

–p
: (parent) mkdir creates any non-existing parent directories that are given in the path name of *directory* before creating the directory itself. The parent directories get the standard access permissions plus u+wx to grant the creation of subdirectories.

––
: If *directory* begins with a dash (–), the end of the command-line options must be marked with ––.

directory
: Name of the directory that you wish to create. You can also create more than one directory at a time.

directory can be given either as a relative path name or as an absolute path name.

The user ID and the group ID of the new directory are set to the real user ID and real group ID of the calling process.

EXIT STATUS

0
: If all directories given in the command line were made successfully.

>0
: An error occurred. mkdir also prints an error message.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Creating a new directory named letters in the directory /home/sisyphus/general:

First you check which directory you are in. The default directory mode is set to 777.

```
$ pwd
/home/sisyphus
```

Next you list the contents of this directory:

```
$ ls -l
total 145
-rw-r--r--   1 sisyphus   group1    5329   Nov 03 09:54 diff.rc.1
drwx--x--x   2 sisyphus   group1    2589   Aug 03 15:08 general
   .
   .
   .
drwxr-----   3 sisyphus   group1    2340   Jun 11 15:35 lingua
drwx------   2 sisyphus   group1    3380   Oct 11 15:36 post
drwx--x--x   2 sisyphus   group1    2080   Nov 04 16:08 proc
drwx------   2 sisyphus   group1    1560   Oct 11 15:36 screens
```

Then you create the new directory:

```
$ mkdir general/letters
```

Finally you check whether a directory named `letters` has been created:

```
$ cd general
$ ls -l
total 5
drwx--x--x   2 sisyphus   group1     520   Jan 22 16:21  letters
```

SEE ALSO

`rm`(1), `rmdir`(1), `sh`(1), `umask`(1), `mkdir`(2).

User's Reference Manual

mkfifo(1)

NAME

mkfifo - make FIFO special file

SYNOPSIS

mkfifo [–m *mode*] [––] *file* ...

DESCRIPTION

mkfifo creates the FIFOs special files specified in its argument list.

For every *file* argument, the mkfifo command behaves as if the mkfifo(3C) was called with the following arguments:

- the *path* argument of mkfifo(3C) is set to *file*;
- the *mode* argument of mkfifo(3C) has the value 0666 or the value of *mode* if the –m option is specified.

If an error occurs during the creation of a FIFO special file, mkfifo writes a diagnostic message to the standard error output and then continues with the remaining arguments, if any.

OPTIONS

No option specified:
: mkfifo creates the FIFO special files specified in *file* with the access rights 666, modified by the current file-creation mode mask [see umask(1)].

–m *mode*
: mkfifo creates the new FIFO special file with the access rights specified in *mode* [see chmod(1)]. If specified symbolically, the operands + and – are interpreted relative to a default value of a=rw.

––
: End of the list of options. Must be specified if *file* begins with –.

file
: The name of the FIFO special file that you would like to create. You can specify a number of FIFOs.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

SEE ALSO

chmod(1) , umask(1) , mkfifo(3C) .

User's Reference Manual

mkmsgs(1)

NAME

mkmsgs - create message files for use by gettxt (makemessages)

SYNOPSIS

mkmsgs [–i *locale*] [–o] *input_file msgfile*

DESCRIPTION

mkmsgs creates a file of text strings that can be accessed by the text retrieval tools gettxt(1), srchtxt(1) and exstr(1) and the C function gettxt(3C). The input to mkmsgs consists of an editable file of text strings in a language based on a particular geographic locale [see setlocale(3C)]; the output file contains the same text strings, but in a format that can be retrieved by both gettxt and gettxt(3C). mkmsgs sequentially transforms and copies the strings from the input file to the output file.

OPTIONS

No option specified:
msgfile is installed in the current directory.

–i *locale*
The –i option is intended for the system administrator only.

locale represents the native language in which the strings in *input_file* are written. The value given to *locale* must subsequently be assigned to the environment variable LC_MESSAGES or LANG so that the file can be accessed by gettxt(3C) (see **EXAMPLES**). A message file named *msgfile* which gettxt(1) and gettxt(3C) can read will then be created in the directory /usr/lib/locale/*locale*/LC_MESSAGES, where your specified *locale* defines the basename of the directory /usr/lib/locale/*locale*. Only the system administrator or a member of the group bin can create or overwrite files in this directory. If the directory which you name as *locale* does not exist under /usr/lib/locale, it will be created.

Non-privileged users are not permitted to create directories under /usr/lib/locale or files under /usr/lib/locale/*locale*/LC_MESSAGES.

–o *msgfile* will be overwritten if it already exists.

input_file
File containing the original text strings that mkmsgs is to convert. Each string must be entered in a separate line. Non-graphic characters must be represented as alphabetic escape sequences, such as \t for a tab or \n for a newline (as with the printf(3S) function in C). Octal representations in the form *nnn* are also permitted. The null string can be represented as an empty line.

Strings can be changed by editing the file *input_file* with any editor. Existing strings may be overwritten, but new strings must be added to the end of the file. If you modify *input_file*, you must create a new *msgfile* file and install it at the correct position in the file tree, i.e. under /usr/lib/locale/*locale*/LC_MESSAGES. This allows the gettxt command or the text retrieval function gettxt(3C) to access the message file when LC_MESSAGES (or LANG if LC_MESSAGES is undefined) is set to the value of *locale*. If this procedure is not followed, the gettxt(3C) function will retrieve the wrong string, and software compatibility between different versions will be broken.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The following strings appear in the input file `C.str`:

```
File %s:\tcannot be opened\n
%s: Bad directory\n
  .
  .
  .
Write error\n
  .
  .
```

The following invocation of `mkmsgs` uses the input strings from `C.str` to create text strings in the appropriate format in the message file `UX` in the current directory:

```
$ mkmsgs C.str UX
```

This next invocation uses the input strings from `FR.str` to create text strings in the appropriate format in the file `UX` in the directory `/usr/lib/locale/french/LC_MESSAGES`:

```
$ mkmsgs -i french FR.str UX
```

If you set the environment variable `LC_MESSAGES=french`, the text strings in `/usr/lib/locale/french/LC_MESSAGES/UX` can be accessed with the `gettxt` command or the C function `gettxt(3C)`.

Example 2

The following files are present:

Contents of `input_file1`:

```
    Hallo !\n\007
    Gute\t... Nacht !\n
    Guten Morgen !\n
```

Contents of `input_file2`:

```
    Salut !\n
    Bonne nuit !\n
```

The message files are created and installed as follows:

The directories `/usr/lib/locale/german/LC_MESSAGES` and `/usr/lib/locale/french/LC_MESSAGES` do not exist. The system administrator therefore gives the following command to create the directory `/usr/lib/locale/german` with a file named `hallo` in it.

```
$ mkmsgs -i german input_file1 hallo
```

The command:

```
$ mkmsgs input_file2 hallo
```

creates a file named `hallo` in the current directory. In order to use this file later in the `french` locale, the system administrator must perform the following actions:

```
$ mkdir /usr/lib/locale/french
$ mkdir /usr/lib/locale/french/LC_MESSAGES
$ cp hallo /usr/lib/locale/french/LC_MESSAGES/hallo
```

These files are accessed as follows:

If `LC_MESSAGES` is empty and the value of `LANG` is `german`:

```
$ gettxt hallo:1 "Hello!\n"
Hallo !
```

The terminal bell also rings at the same time.

```
$ gettxt hallo:2 "Good Night!\n"
Gute   ... Nacht !
$ gettxt hallo:3 "Good Morning!\n"
```

```
Guten Morgen !
$ LC_MESSAGES=french
$ export LC_MESSAGES
$ gettxt hallo:1 "Hello!\n"
Salut !
$ gettxt hallo:2 "Good Night!\n"
Bonne nuit !\n
$ gettxt hallo:3 "Good Morning!\n"
Message not found!!
```

The call featuring `hallo:3` causes `Message not found!!` to be displayed, since the file `/usr/lib/locale/french/LC_MESSAGES/hallo` only contains 2 messages.

```
$ LC_MESSAGES=invalid
$ gettxt hallo:1 "Hello!\n"
Hello
$ gettxt hallo:2 "Good Night!\n"
Good Night!
```

The calls featuring `hallo:1` or `hallo:2` print the specified text, as there is no file named `/usr/lib/locale/invalid/LC_MESSAGES/hallo`. If no text were specified, the string "`Message not found`" would be displayed [see `gettxt(1)`].

FILES

`/usr/lib/locale/*/LC_MESSAGES/*`

Message files created by `mkmsgs`. There is at least one directory under `/usr/lib/locale` for each supported language.

SEE ALSO

`exstr(1)`, `gettxt(1)`, `srchtxt(1)`, `gettxt(3C)`, `setlocale(3C)`.

International environment (NLS locale).

User's Reference Manual

more(1)

NAME

more - control the output on the screen

SYNOPSIS

more [–[+]aBcCdeEfFHiIlLmMqQrsuUw] [–b*number*] [–j*number*] [–[n]*number*] [–k*number*] [–I*number*] [–P[mM=]*string*] [–o *file* | –O *file*] [–K*file*] [–t*tag*] [–T*tagsfile*] [–p*command*] *file ...*

more –h

DESCRIPTION

more reads files and outputs them page-by-page or to standard output. It allows both forward and backward movement in a file. more does not have to have read the entire input file before starting, and therefore starts up faster than text editors such as vi(1) , for example.

more uses the terminfo(4) file and can therefore run on a variety of terminals. There is also limited support on hardcopy terminals. (On a hardcopy terminal, lines which should be displayed at the top of the screen are prefixed with an up-arrow.)

OPTIONS

Command line options are described below. Most of the options can be changed with the – command while more is active.

Options are also read from the MORE environment variable. If, for example, you do not want to enter more –*options* ... for every more call, you can enter the following for the C shell csh.

setenv MORE –*options*

If you use the sh Bourne shell, enter the following:

MORE=–*options*; export MORE

The environment variable is evaluated before the command line, so that the command line options can overwrite the value of the MORE environment variable. If the MORE variable contains an option, you can reset the option in the command line to its default value by preceding the command line option with –+.

The dollar sign ($) indicates the end of an option value. This is only important for options such as –P that are followed by a string.

–a Searches are started after the last line displayed on the screen, i.e. all lines displayed on the screen are skipped. By default, searches start in the fourth line of the display (or after the last line found, see option –x).

–b*number*
Causes more to use a number of buffers that does not correspond to the default number. The buffers are each 1 KB in size, and 10 buffers are used by default (except if the data is coming from the standard input; see option –B). *n* specifies the number of buffers to be used.

–B Disables the automatic allocation of buffers so that the default number of buffers is used. If more data is read than can fit in a buffer, that oldest data is deleted. If data comes from the standard input, the buffers are by default automatically allocated as required, to avoid data loss.

–c The entire screen is refreshed from top to bottom. This happens by default when scrolling from the bottom of the screen.

–C The –C option behaves like –c, but the screen is deleted before being refreshed.

–d The –d option suppresses error messages that would normally be displayed on a non-intelligent terminal, i.e. if the terminal was missing an important function such as the facility to clear the screen or scroll backward. The –d option otherwise has no influence on the behavior of more on a non-intelligent terminal.

`-e` Causes `more` to automatically exit the file the first time it reaches the end of the file. By default, `more` exits when it reaches the end of the file for the second time.

`-E` Causes `more` to quit after `q` is entered, not automatically.

`-f` Lines that are longer than the screen width are not folded, the part of the line that cannot be displayed is truncated. This part of a long line is simply deleted. By default, long lines are folded, i.e. the part of the line which goes beyond the width of the screen is displayed in the next line.

`-F` Forces non-regular files to be opened. (A non-regular file is a directory or a device-oriented file.) This option also suppresses the warning message when opening a binary file. By default, `more` will not open a non-regular file.

`-h` This option shows a summary of the commands accepted by `more`. (Corresponds to the `h` command). If this option is specified, all other options are ignored. In this case, `more` exits after the help screen is displayed.

`-i` Case is ignored during a search, i.e. lowercase and uppercase are regarded as the same. A search can also be conducted for overscored or underscored text. This option is ignored if the search pattern itself contains uppercase letters.

`-I`*number*
A tab is set after each *n* columns. The default value for *n* is `8`.

`-j`*number*
Specifies the maximum number of lines can that be scrolled forward. If more than *n* lines must be scrolled forward, the screen is refreshed instead. With the `-c` or `-C` option, the screen can be refreshed from top to bottom if necessary. By default, every forward movement is a scrolling process.

`-k`*n* Specifies the maximum number of lines that can be scrolled backward. If more than *n* lines must be scrolled forward, the screen is instead refreshed in a forward direction. (If the terminal cannot scroll forward, `-k0` is accepted implicitly.)

`-K`*file*
`more` opens and interprets the specified file as a `lesskey(1)` file. The `-K` option can be specified more than once. If there is a `.more` in your `HOME` directory, this file is also used as a `lesskey` file.

`-l` Suppresses line numbers. By using the default value (using the line number), `more` can be extremely slow in some cases, especially if the input file is large. This problem is avoided by suppressing line numbering with the `-l` option. The use of line numbering means that the line number is displayed in the detailed prompt and in the `=` command. The `v` also passes the current line number to the editor. (Read the explanation of `LESSEDIT` in the **PROMPTS** section below.)

`-L` Displays a line number at the start of each line on the screen.

`-m` Causes `more` to output only a colon a prompt. By default, the prompt for `more` is the filename and the percent into the previous file.

`-M` Causes `more` to prompt in a more detailed manner.

`-[n]`*number*
changes the default value for scrolling *n* lines. The default value is a screenful. With the `z` and `w` commands, the window size can also be modified. The `n` character can be left out of older versions of `more` for compatibility reasons (example: `-`*n*).

`-o`*file*
Causes `more` to copy the input in the specified file while it is displayed. This happens only if the input file is a pipe and not a normal file. If the file already exists, `more` asks you to confirm the overwriting of the file.

`-O`*file*
The `-O` option corresponds to the `-o` option, except that an existing file is overwritten without a confirmation request.

If there is no log file specified, it can be specified by `more` using the `-o` and `-O` options. If there is no

filename specified for these options, they will transfer only the name of the log file. The `o` command corresponds to specifying `-o` within `more`.

`-p`*command*

The `-p` option in the command line is the same as specifying `+/`*command*, i.e. `more` executes *command* each time at new file is displayed.

This is the same as the `+`*command* that is also supported.

`-P`*prompt*

Allows the three prompt forms to be customized to meet your requirements. This option is normally entered in the `MORE` environment variable and not re-entered for every new `more` command. Such an option must either be specified as the last option in the `MORE` variable or be terminated with a dollar sign. If `-P` followed by a string is specified, the standard medium prompt in this string is changed. `-Pm` changes the short prompt (`-m`) in the specified string, and `-PM` changes the long prompt (`-M`). `-P=` also changes the message returned by `=` in the specified command line. All strings for a prompt consist of a combination of letters and special escape sequences. You can find more information in the **PROMPTS** section.

`-q` The process is executed relatively "quietly": the terminal does not signal acoustically if an attempt is made to scroll past the beginning or end of the file. If the terminal has a "visual signal" facility, this is used instead of the acoustic signal. The acoustic signal sounds for certain other errors, e.g. if an invalid character is entered. By default, the terminal sounds in all cases.

`-Q` The process is executed completely "quietly": The acoustic signal is never sounded.

`-r` Control characters are displayed in RAW mode (usually unchanged). By default, control characters are displayed using the carat notation. For example, Ctrl+`A` (Octal 001) is represented as `^A`.

Warning: If the `-r` option is used, `more` cannot follow the screen output, because it depends on how the screen reacts to each type of control character. Various screen problems may result, such as long lines being split in the wrong place.

`-s` Consecutive blank lines are squeezed into one blank line. This is useful when displaying `nroff(1M)` output.

`-t`*tag*

This command processes the file containing the specified *tag* mark. It only works if the current directory contains a `tags` file created using the `ctags(1)` command. By specifying this option within `more` (`:` command), the corresponding file is displayed. The `:t` command is the same as specifying `-t` within `more`.

`-T`*tagsfile*

Specifies a "tags" file that should be used instead of `tags`.

`-u` Backspaces and carriage returns are treated as printable characters, i.e. these characters are sent to the terminal if they appear in the input.

`-U` Backspaces and carriage returns are treated as control characters, i.e. they are dealt with as specified by the `-r` option.

If neither `-u` nor `-U` is specified, backspaces adjacent to an underscore have a special meaning: The underscored text is displayed using the terminal hardware's underscoring facility. Backspaces between two of the same characters also have a special meaning: The overstruck text is displayed using the terminal hardware's boldface facility. Other backspaces are deleted along with any preceding characters. Carriage returns immediately followed by a NEWLINE character are deleted. All other carriage returns are dealt with as specified by the `-r` option.

`-w` Lines after the end of the file are represented as blank lines. By default, a tilde is used for this.

`-x`*number*

Specifies a line on the screen where "target" lines should be positioned. Target lines are the result of text or marks searches, jumps to a line, to a position specified by a percentage, or to a marked position. The screen line is specified by a number: The first line is number 1, the second number 2 and

so on. A line relative to the bottom line of the screen is specified by a negative number: The last line on the screen is number −1, the second- last number −2 and so on. If the −x is specified, the search begins immediately after the target line. With −x4, for example, the target line is the fourth line on the screen and the search begins in the fifth line.

+ If an option in the command line begins with +, it is interpreted as an initial command for more. With +G, for example, more is instructed to begin at the start of the file and not the end. +/xyz instructs more to start at the first occurrence of "xyz". +*number* is a special case and behaves like +*number*g, i.e. the display is started with the *number* line. (Note the g command below.) If the option begins with ++, the initial command is valid for every file displayed, and not just for the first. Using the + command, an initial command can be set (or changed) for every file.

file A pathname of an input file.

COMMANDS

The commands are based on the vi editor. The commands can be preceded by a decimal number, described as *N* in the descriptions below. The effects of *N* are described in the individual commands.

In the following descriptions, ^X stands for Ctrl+X. ESC stands for the ESCAPE key; e.g. ESC-v stands for the key sequence "ESCAPE" and "v".

h or H
: Help: Displays a summary of all the commands. This is the most important command to know.

SPACE or ^V or f or ^F
: Scrolls forward *N* lines. The default value is one window (see the −n option above). If *N* is greater than the number of lines in a window, the *N*th line of the last screen is displayed.

  Warning: In some systems the ^V character is used as a metacharacter to represent non-printing characters.

z
: Like SPACE. If *N* is specified, this specifies the new window size.

RETURN or ^N or e or ^E or j or ^J
: Scrolls forward *N* lines. The default value is one line. The entire *N* lines are shown, even if the number of lines is greater than the size of the screen.

  At the file end RETURN has the effect that more displays the next file from the list of files, or quits, if the current file was the last one.

s
: Skips the next *N* lines and displays the next screen. The default value for *N* is one line. If after the skip there are less lines than fit on a screen, a full screen is displayed nonetheless with the end of the file.

d or ^D
: Scrolls forward *N* lines. The default value is half the screen. If *N* is specified, this value becomes the new default value for the d and u commands that follow.

b or ^B or ESC-v
: Scrolls backward *N* lines. The default value is a screen (see the −n option above). If *N* is greater than the number of lines on a screen, the *N*th line of the last screen is displayed.

w
: Like b. If *N* is specified the new window size to be scrolled through is defined.

y or ^Y or ^P or k or ^K
: Scrolls backward *N* lines. The default value is one line. The entire *N* lines are shown, even if the number of lines is greater than the size of the screen.

  Warning: In some systems, the ^Y is used as a metacharacter for job sequencing.

u or ^U
: Scrolls backward *N* lines. The default value is one half of the screen. If *N* is specified, this value becomes the new default value for subsequent d and u commands.

r or ^R or ^L
: Refreshes the screen.

`R` Refreshes the screen and deletes any input saved in the buffer. This command is useful if the file changes while it is being displayed.

`F` Scrolls forward and keeps trying to read the file when it reaches the end. This command is normally when already at the end of the file. This means the end of the file that is growing while it is being displayed. (The behavior of the command is similar to that of the `tail –f` command.)

`g` or `<` or ESC-`<`
Goes to line *N* in the file. The default value is the beginning of the file.

Warning: This command can take some time if a large value is specified for *N*.

`G` or `>` or ESC-`>`
Goes to line *N* in the file. The default value is the end of the file.

Warning: This command can take some time if a large value is specified for *N*, or if *N* is not specified, and the standard input is read instead of a file.

`p` or `%`
Goes to a position *N*% into the file. *N* must be a value between 0 and 100. (This also works if the standard is read, but only if `more` has already read the file to the end. This command executes quickly, but is not always useful.)

`{` If there is a left brace is the top line of the screen, the `{` command goes to the corresponding right brace. This brace is displayed in the bottom line of the screen. If there is more than one left brace in the top line of the screen, a number *N* can be used to specify the corresponding right brace.

`}` If there is a right brace in the bottom line of the screen, the `}` command goes to to the corresponding left brace. This brace is displayed in the top line of the screen. If there is more than one right brace in the bottom line of the screen, a number *N* can be used to specify the corresponding left brace.

`(` Like `{`, but this command applies to parentheses rather than braces.

`)` Like `}`, but this command applies to parentheses rather than braces.

`[` Like `{`, but this command applies to square brackets rather than braces.

`]` Like `}`, but this command applies to square brackets rather than braces.

ESC-`^F`
This command behaves like `{` when followed by two characters, however it uses both characters as open and close brackets. With the ESC+`^F < >` command for example, you can scroll forward to the `>` character. This character corresponds to the `<` character in the top line of the screen.

ESC-`^B`
This command behaves like `}` when followed by two characters, however it uses both characters as open and close brackets. With the ESC+`^B < >` command for example, you can scroll backward to the `<` character that corresponds to the `>` character in the bottom line of the screen.

`m` A lower case letter is specified after this command. This command marks the current position with this letter.

`'` or `^X^X`
(Single quote or `^X^X`) Followed by any lowercase letter, this command returns to the position previously marked with the specified lower case letter using the `m` command. If this command is entered with another single quote, it goes back to the position at which the last "large" movement command was executed. If this command is followed by the `^` or `$` character, it skips to the beginning or the end of the file. Markings are preserved when a new file is displayed so the `'` command can be used to switch between files.

/pattern
Searches forward in the file for the *N*th line containing the specified pattern. The default value for *N* is `1`. The pattern is a regular expression, as in the `ed` editor. The search begins in the fourth line displayed. (Read the descriptions of the `–a` and `–x` options, which alter the behavior of this command.)

Some characters have a special meaning when specified at the start of a pattern. They change the search type and do not belong to the search pattern. This applies to the following characters:

`!` Searches for lines that do NOT match the search pattern.

`*` Searches multiple files: if the search reaches the end of the current file without finding the specified pattern, the search is continued in the next file specified in the command line.

`@` The search is started in the first line of the first file specified in the command line. This occurs regardless of the current screen displayed and the settings of the `-a` and `-x` options.

`?`*pattern*
Searches backward in the file for the *N*th line containing the specified pattern. The search starts in the line immediately before the current position (i.e. in the second line displayed).

As in the `/`*pattern* command, some characters in this command have a special meaning:

`!` Searches for lines that do NOT match the search type.

`*` Searches multiple files: If the search reaches the beginning of the current file without finding the specified pattern, the search is continued in the previous file specified in the command line.

`@` The search is started in the last line of the last file specified in the command line. This occurs regardless of the current screen displayed and the settings of the `-a` and `-x` options.

ESC-`/`*pattern*
Like `/*`.

ESC-`?`*pattern*
Like `?*`.

`n` Repeats the last search process and searches for the *N*th line containing the last pattern specified. If the last search command was modified by `!`, the search is made for the *N*th line that does NOT contain the specified pattern. If the last search command was modified by `*`, the search process is continued in the next (or preceding) file, provided that the pattern is not found in the current file. If the last search command was modified by `@`, `n` has no effect.

`N` Repeats the last search, but in the opposite direction.

ESC-`n`
Repeats the last search process across file boundaries. The effect is the same as if the last search command were modified by `*`.

ESC-`N`
Repeats the last search, but in the opposite direction, and across file boundaries.

`:e` [*file*] or `E` [*file*] or `^X^V` [*file*]
Displays a new file. If no filename is specified, the "current" file (see the descriptions of the `:n` and `:p` commands below) from the list of files in the command file is displayed again. A percent sign (`%`) in the filename is replaced by the name of the current file. A number sign (`#`) is replaced by the name of the last file displayed. The filename is added to the list of files in the command line so that it is considered by subsequent `:n` and `:p` commands. If a number of files are specified under *file*, all the files are added to the list and the file specified first is displayed.

Warning: In some systems the `^V` is used as a metacharacter to represent non-printing characters.

`:n` Displays the next file in the command line. If a number *N* is specified, the *N*th next file is displayed.

`:p` Displays the previous file in the command line. If a number *N* is specified, the *N*th-last file is displayed.

`:x` Displays the first file in the command line. If a number *N* is specified, the *N*th file in the list is displayed.

`=` or `^G` or `:f`
Outputs information about the specified files. This information includes the filename, the line number, and the byte offset from the bottom line. If possible, the length of the file, the number of lines, and the percentage of the file above the last line displayed is also output.

`-` In combination with one of the command line option letters (see above), this command changes the setting of the corresponding option and returns a message describing the new setting. If the option

letter has a numeric argument (such as –b or –k) or is specified with a string (such as –P or –t), a new value for this argument can be entered after the option letter. If no new value is entered, a message with the current setting is output, and this setting is not changed.

–+ In combination with one of the command line option letters (see above), this command resets the corresponding option to its default value and returns a message describing the new setting. (The –+*X* command behaves in exactly the same manner as –+*X* in the command line.) This does not function for options that expect strings as arguments.

–– In combination with one of the command line option letters (see above), this command resets the corresponding option to the opposite of its default value and returns a message describing the new setting. (The ––*X* command behaves in exactly the same manner as the –*X* in the command line.) This does not function for options that expect strings as values.

_ (Underscore) In combination with one of the command line option letters (see above), this command returns a message with the current setting of the specified option. The option setting is not changed.

+*command*
The *command* command is executed every time a new file is displayed. By specifying +G, for example, more displays every file from end to beginning, and not from beginning to end.

V Outputs the version number of the active more program.

q or :q or :Q or ZZ or ESC+ESC
Exits more.

v Calls an editor to process the file just displayed. The editor to be called up is extracted from the EDITOR environment variable. If no editor is specified there, vi is used by default. Read the explanations of LESSEDIT in the **PROMPTS** section below.

! *shell-command*
Calls a shell that executes the shell command specified by *shell-command*. A percent sign (%) in the command is replaced by the name of the current file. A number sign (#) is replaced by the name of the last file displayed. !! repeats the last shell command. ! without a shell command calls a shell. In all cases the shell to be called is extracted from the SHELL environment variable or the sh default value is used.

| *m shell-command*
m represents a mark letter. A section of the input file is piped to the specified shell (*shell-command*). The section is between the current position and the position indicated by the mark letter. *m* can have a value of ^ or $, to specify the beginning or end of the file. If *m* is . or if a NEWLINE character is specified, the current screen is piped. The current screen is the smallest part of the file that can be piped.

KEY BINDINGS

You can define some more commands by creating a .more file with the lesskey program in your HOME directory. This file specifies a set of command keys and actions allocated to the individual keys. You can find more information about this under lesskey(1) .

NATIONAL CHARACTER SETS

The input file contains three types of characters:

Normal (printable) characters
can be displayed directly on the screen.

Control characters
should not be displayed directly, but can be contained in normal text files (e.g. backspaces and tabs).

Binary characters
cannot be displayed directly and should not normally be contained in text files.

By default, more uses the national environment character set (defined by the LC_ALL, LC_TYPE, and LANG environment variables). With the LESSCHARSET environment variable, a different character set can be

selected. If this variable is set to `ascii`, the ASCII character set is used. In the ASCII character set, characters with values between 128 and 255 are interpreted as binary characters. If this variable is set to `latin1`, the ISO 8859/1 character set is used. The Latin-1 character set is the same as the ASCII character set with the exception that characters between 128 and 255 are interpreted as normal characters. If this variable is set to `locale`, the national character set defined in the local environment is used. At the moment only `locale`, `ascii` and `latin1` values are valid for `LESSCHARSET`.

In some cases it may be desirable to change `more`, so that a different character set is used to the one defined in `LESSCHARSET`. To do this a character set is defined in the `LESSCHARSET` environment variable. This variable should be allocated a string that describes every character in the set. `.` is used to indicate normal characters, `c` control characters, `b` binary characters, and `l` for the character type in the local environment. A decimal number can be specified to repeat this. Specifying `bccc4b.`, for example, means that 0 is a binary character, 1, 2, and 3 are control characters, 4, 5, 6, and 7 are binary characters and 8 is a normal character. The specification for the last character is valid for all characters following it, i.e. 9-255 are normal characters. (This is only an example and does not necessarily represent an actual character set.)

Setting the `LESSCHARDEF` variable to `8bcccbcc18b95.b` is the same as setting `LESSCHARSET` to `ascii`. If you set the `LESSCHARDEF` variable to `8bcccbcc18b95.33b.`, it is the same as setting `LESSCHARSET` to `latin1`. If you set the `LESSCHARDEF` variable to `8lccclccl`, it is the same as setting `LESSCHARSET` to `locale`.

Control and binary characters are displayed blinking. Each of these characters is indicated by a carat, if possible (e.g. `^A` for Ctrl+`A`). This indication is only used when converting the 0100 bit to a normal printable character. Otherwise the character is represented as an octal number preceded by a backslash. You can change this octal format by setting the `LESSBINFMT` to a format string as in `printf`. The default value is `'\%o'`.

PROMPTS

With the `-P` option you can customize the prompt to suit your needs. The prompt is replaced by the string specified in the `-P` option. Certain characters in the string are interpreted in a particular way. The prompting mechanism is very complicated to be able to offer flexibility in the way the prompt is formulated.

A combination of percent signs followed by a single character is expanded to what the following character is:

`%b`*X* Is replaced by the byte offset of the current input file. `b` is followed by a single character (as in *X* above), specifies the line whose byte offset should be used. If this character is `t` (top), the byte offset of the top line of the display is used, `m` (middle) indicates the middle line, `b` (bottom) the bottom line, and `B` the line directly after the bottom line. `x` specifies that the "target line" specified by the `-x` option should be used (as in `-x`).

`%B` Is replaced by the size of the current input file.

`%E` Is replaced by the name of the editor (from the `EDITOR` environment variable, otherwise `vi`). You can find more information about this in the description of the `LESSEDIT` environment variable below.

`%f` Is replaced by the name of the current input file.

`%i` Is replaced by the index of the current file in the list of input files.

`%l`*X* Is replaced by the number of a line in the input file. The line to be used is specified in the `%b` option under *X*.

`%L` Is replaced by the number of the last line in the input file.

`%m` Is replaced by the number of input files.

`%p`*X* Is replaced by the percentage proportion of the current file. The line to be used in specified in the `%b` option under *X*.

`%s` Like `%B`.

`%t` Subsequent blanks are deleted. This option is normally specified at the end of a string, but can occur anywhere.

`%x` Is replaced by the name of the next input file in the list.

If there is an unknown element (e.g. the file size, if the input is a pipe), a question mark is used instead.

The string format for the prompt can be changed depending on certain conditions. A question mark followed by a single character, behaves like an `IF` instruction. Depending on the character after the question mark, a condition is evaluated. If this condition is TRUE, all characters after the question mark and condition character, up to a period, are included in the prompt. If the condition is FALSE, the characters are not included. Specifying a colon between the question mark and the period is the same as an `ELSE` instruction: the character between the colon and the period are only added to the prompt if the `IF` condition is FALSE. The following characters can be used as condition characters (after a question mark):

`?a` True if characters have already been included in the prompt.

`?b`*X* True if the byte offset of the specified file is known. Like `%b`, *X* can have the values `t` (top), `m` (middle) and `b` or `B` (bottom). `x` as well.

`?B` True if the size of the current input file is known.

`?e` True if the end of the file is reached.

`?f` True if the input file is named, signifying input and not a pipe.

`?l`*X* True if the number of the specified line is known. Like `%b`, *X* can have the values `t` (top), `m` (middle) and `b` or `B` (bottom). `x` as well.

`?L` True if the number of the last line in the file is known.

`?m` True if there is more than one input file.

`?n` True if it is the first prompt in a new input file.

`?p`*X* True if the percentage proportion of the file up to the specified line is known. Like `%b`, *X* can have the values `t` (top), `m` (middle) and `b` or `B` (bottom). `x` as well.

`?s` Like `?B`.

`?x` True if there is another input file, i.e. the current file is not the last one.

All characters, with the exception of metacharacters (question marks, colons, periods, percentage signs and backslashes), are prompt elements. If metacharacters are also to be included in the prompt, they must be preceded by a backslash.

Some examples:

```
?f%f:Standard input.
```

This prompt outputs the name of the file if it is known. If the name is not known, the `Standard input` string is output.

```
?f%f .?ltLine %lt:?pt%pt\%:?btByte %bt:-...
```

This prompt outputs the filename if it is known. The line number, if known, is then output. If the line number is not known, the percentage information is output, and if this is not known either, the byte offset is output. If no value is known, a dash is printed. Note that for every question mark there is a corresponding period, and the `%` special character is added after `%pt` in the prompt, because it is preceded by a backslash.

```
?n?f%f□.?m(file□%i□of□%m)□..?e(END)□?x-□Next\:□%x.:
        ?pB%pB\%:byte□%bB?s/%s...%t
```

The strings are divided into two lines to improve legibility. If the prompt is the first in a file, the filename is output. A `(file` *N* `of`*N*`)` message is output after the filename if there is more than one file. At the end of the file the `(END)` string is output followed by the name of the next file, if one exists. Otherwise percentage output information is output. If this is not known, the byte offset is output, followed by the file size (if known). Subsequent null characters are then truncated. This is the standard prompt.

The default values for the two other prompts (`-m` or `-M`) are shown for reference purposes.

```
?n?f%f□.?m(file□%i□of□%m)□..?e(END)□?x-□Next\:□%x..%t
```

```
?f%f□.?n?m(file□%i□of□%m)□..?ltline□%lt?L/%L.□:byte□%bB?s/%s.□.
        ?e(END)□?x-□Next\:□%x.:?pB%pB\%..%t
```

The `=` outputs the following default message:

```
?f%f□.?m(file□%i□of□%m)□.?ltline□%lt?L/%L.□.
        byte□%bB?s/%s.□?e(END)□:?pB%pB\%..%t
```

The prompt expansion features are also used for another purpose: if a LESSEDIT environment variable is defined, its value is executed as the v command. The string in LESSEDIT is expanded in the same way as the prompt strings. LESSEDIT has the following default values:

```
%E□?lx+%lx.□%f
```

This value expands to the editor name, followed by a +, the line number and the filename. If your editor does not accept the *+linenumber* syntax or if there are other discrepancies in the syntax for calling the editor, you can change the default value by processing the LESSEDIT variable.

ENVIRONMENT VARIABLES

COLUMNS
: Determines the number of columns on the screen. This value has priority over the number of columns set in the TERM variable.

EDITOR
: The editor name (used for the v command).

HOME
: The name of the user's HOME directory (used for searching for the .more file).

.TP
: LANG Specifies a default value for the internationalization variables that are unset or null. If LANG is unset or null, the corresponding default value from the international environment is used. If the internationalization environment contains an invalid setting, the command behaves as if none of the variables were set.

LC_ALL
: If this variable is set, i.e. is not empty, this value overwrites the values of all other internationalization variables.

LC_COLLATE
: Determines the international environment for the behavior of ranges, equivalence classes, and character units within the regular expressions.

LC_CTYPE
: Determines the international environment for the interpretation of byte sequences as characters (e.g. as single-byte as opposed to multibyte characters in arguments) and for the behavior of character classes within regular expressions.

LC_MESSAGES
: Determines the format and content of error messages. The international environment specified here also applies for informative messages written to the standard output.

NLSPATH
: Determines the position of the message catalog for the processing of LC_MESSAGES.

LESSBINFMT
: The format for displaying non-printing characters that are not control characters.

LESSCHARDEF
: Defines a character set.

LESSCHARSET
: Selects a predefined character set.

LESSEDIT
: Editor string (is used for the v command). You can find more information in the **PROMPTS** section.

LINES
: Determines the number of lines on the screen This value has priority over the number of lines determined by the TERM variable.

MORE
: Parameters automatically transferred to more.

SHELL
: The shell for the execution of the ! command as well as for the expansion of file names.

`TERM` The type of terminal on which `more` is being executed.

WARNINGS

If a number of files are specified with `:e`, and one of these files was already displayed, these new files may be entered in an unexpected order in the list.

The handling of national character sets is both non-standard and insufficient for multibyte characters (unless the default value or `locale` is used). This may change in a later release.

SEE ALSO

`less(1)`, `lesskey(1)`.

User's Reference Manual

mt(1)

NAME

`mt` - tape control utility

SYNOPSIS

`mt` [`-f` *tapename*] *command* [*count*]

DESCRIPTION

You can use `mt` to send commands to the tape drives, in order, for example, to

- retain a streamer tape,
- rewind a tape/streamer tape to the beginning,
- wind forward to a specific position,
- write an end-of-archive mark,
- extend the reservation of a tape drive, so that it remains reserved for the initiator when the job has been completed,
- abort the job on a tape drive,
- print status information about the tape unit.

OPTIONS

`-f` *tapename*

Addresses the driver of the named unit. `mt` interprets the next argument as the name of a special file.

tapename

Here you specify the special file for the device you wish to address. The special file must be a file for a raw device; it must on no account be a block special file [see `stape(7)`].

The name of the special file determines, for example, whether or not the tape is to be wound back automatically to the start following access.

To transport a tape to a specific position with `mt` you have to select the special file which does not cause the tape to be rewound.

`-f` *tapename* not specified:

`mt` reads the `TAPE` environment variable.

This enables you to use a predefined special file. If you assign the name of your chosen special file to the environment variable `TAPE`, you can then call `mt` without using the `-f` option and `mt` will access the predefined special file automatically.

command

command which the designated driver is to execute. Some commands have two different forms, and in these cases both versions of the command are shown before the description of the function.

The available commands are as follows:

`fsf` Wind on to the start of the next archive.

An archive consists of all files between:

- the start-of-tape mark and the first end-of-archive mark or
- two end-of-archive marks.

The last archive on the tape is bounded by two end-of-archive marks.

Commands such as `cpio` and `tar` automatically write an end-of-archive mark on the tape when they close the special file after completing the write operation. Hence an archive always contains the files which you have written to tape or tape cartridge in a single write operation.

You should use `fsf` only with special files which do not automatically wind the tape back to the

start following access (see the `-f` option).

`bsf` `bsf` winds the tape back to the previous end-of-archive mark.

You should only use `bsf` with special files which do not automatically wind the tape back to the start following access (see the `-f` option).

`append`

Moves forwards until after the last archive. This command must be used for some tape devices if data is to be added after the last archive.

`fsr` `fsr` winds the tape on to the start of the next record. `mt` does this by looking for the next record gap.

You can use `write(2)` to write data directly to tape. You write a record each time you call `write(2)` , the end of which is marked automatically with a record gap. Records for magnetic tape units can be between 1 byte and 128 Kbytes long.

You should only use `fsr` with special files which do not automatically wind the tape back to the start following access (see the `-f` option) and only with tapes of variable record length.

`bsr` `bsr` winds the tape back to the start of the previous record. `mt` does this by looking for the next record gap. For more information on records, refer to the description of `fsr`.

You should only use `bsr` with special files which do not automatically wind the tape back to the start following access (see the `-f` option) and only for tapes that have been written with variable record length.

`eof`
`weof`

Write an end-of-archive mark at the current tape position. This can be used to create an "empty" archive. A shell script could then read files from the tape until it finds an empty archive.

`rewind`

Wind back to start of tape/streamer tape. The *count* argument, if specified, is ignored.

You should only use `rewind` with special files which do not automatically wind the tape back to the start following access (see the `-f` option).

`offline`
`rewoffl`

This command only works with MTUs.

`offline` and `rewoffl` wind the tape back to the start and place the tape unit off-line. The *count* argument, if specified, is ignored.

Warning:

If the drive cover of a tape cartridge drive is located behind a tape loading door, as on the 8mm magnetic tape cartridge drive of an RM600, the system's tape loading door must be opened before the command is executed. If the door is closed, the drive may well be damaged when opened.

`ret`
`reten`

This command only works with streamers.

`ret` and `retension` retension the tape. The *count* argument, if specified, is ignored.

You should retension the tape in the following instances:

- when you insert a cartridge in the drive;
- after a full tape read or write operation.

`erase`

Erase the tape from the current position to the end of the tape. The *count* argument, if specified, is ignored. On streamers: erase all archives from the tape.

`insert`

Open the drive door. The *count* argument, if specified, is ignored. This is useful for a 8mm drive

within an auto-changer with no tape inserted and drive door closed. The `offline` command will not work in this case.

`status`
: Display status information about the tape unit. The information displayed depends on the type of tape and the type of drive.

  The *count* argument, if specified, is ignored.

  The output always includes the name of the device type used. In addition, the current setting of the buffer size is displayed.

  The output for a streamer on an RM600 e.g. would be:

  ```
  IOS tape drive, residual=8192
  ds=4000<>
  er=0
  ```

  In the top byte, `ds` contains the "Sense Key" and in the lower byte the "Additional Sense Key" from the "Extended Sense Bytes" of the device. `residual` and `er` should not be evaluated.

`reserve`
: Extend the reservation of tape devices. High-performance tape drives often have two independent SCSI ports that can be connected to two systems. Up to now, the tape driver reserved the drive at the start of a job and freed it again when the job was completed. However, if a drive is being used by two systems simultaneously, it can happen that the second system will access the tape before the first system has, for example, finished its data backup. The `reserve` command allows a system to reserve the drive until it frees it again explicitly with a `release` command. If the second system attempts to access the drive while it is still reserved, the device rejects the job with an I/O error and sends a "Reservation Conflict" message to the console. A reservation is only possible if a cassette is inserted in the device. A `reinitdev` command cancels the reservation if it was issued by the previous initiator. A `reinitdev` command from a second system that hadn't reserved it before, is unsuccessful. If the initiator (system) fails for some reason, the reservation can only be canceled by booting the failed system or setting the port on the tape device to "offline"!

  For technical reasons, this cannot be done any other way at present, because the corresponding SCSI commands have not yet been implemented by the manufacturers of the tape devices.

`release`
: Release a tape device that had previously been reserved explicitly with a `reserve` command. This command cancels the extended reservation of a tape device again. The drive can also be freed if no cassette is inserted in it.

  If a `release` or `reserve` command is executed for a tape device, which has already been reserved by another system, the `mt` command issues the following error messages:

  `/dev/ios`*X*`/rstape`*XXX* `Command:` *command* `causes reservation conflict!`

  command, in this case, represents the command `reserve` or `release`.

  A "Reservation Conflict" message is also output on the console by the tape driver.

`relstat`
: Determine the reservation status of a tape drive. If no reservation is effective, the following message is issued:

  `/dev/ios`*X*`/rstape0`*XX* `is not reserved.`

  If a drive was reserved with the `reserve` command by the current system, the following message is issued:

  `/dev/ios`*X*`/rstape0`*XX* `is reserved.`

  If the drive was reserved by another system, the following message is issued:

  `/dev/ios`*X*`/rstape0`*XX* `is reserved by other initiator or system.`

  The reservation messages are also issued, if there is no cassette in the drive, provided one of

the ports is reserved. If the reservation is canceled, the following message is issued when the status is queried:

`Unable to get reservation status of: /dev/ios`*X*`/rstape0`*XX*`Device not read y.`

If this message is issued, it generally means that there is no cassette in the drive. Once a cassette is inserted, the reservation status can be determined again. A different error has occurred if the following message is issued:

`Unable to get reservation status of: /dev/ios`*X*`/rstape0`*XX*

Information regarding the problem that has occurred is output on the system console in this case.

`reset`

The job processed on the tape drive is aborted. This command can only be executed by the system administrator. It can be particularly useful if a tape device requires an unusually long period of time for a command, which would otherwise be completed very quickly, or if a command has been entered inadvertently.

Note:

This command generally results in the tape contents, i.e. the previously saved data, being lost.

`capa` This command is used as follows to display the residual capacity of the cassette inserted in the drive:

`/dev/ios`*X*`/rstape`*XXX*`n: `*NN.NN*` GB of`*NN.NN*` GB (`*PPP*` %) free`

By right, the command should only be used for device files without rewind functionality, as otherwise the cassette will rewind after the command and the total cassette capacity will be displayed again. The results contain an inaccuracy that is dependent on the drive. This cannot be avoided because of the sometime very high storage capacities of the cassettes.

Apart from the DLT(2000 -> 7000) drives, the capacities are always only shown for the normal drive nodes with normal sensitivity.

The DLT drives establish whether the tape was previously written with high sensitivity (node ending is "h") or with normal sensitivity and the residual tape capacity is then output in accordance with the value established. Another special feature to note in the case of DLT drives is that the total capacity of the data medium is not displayed. The following message is issued in this case:

`/dev/ios0/rstape`*XXX*` command: capa -- Total capacity not supported`
`/dev/ios0/rstape`*XXX*`: `*NN.NN*` GB free`

All ExaByte, DAT and DLT drives supported this functionality previously.

count

An integer: *command* is to be executed *count* times. The *count* argument is ignored for the following commands:

`erase`, `offline`, `ret`, `reten`, `rewind`, `rewoffl`, `status`

count not specified:

The specified *command* is executed once.

EXIT STATUS

`0` The driver executed the `mt` command successfully.

`1` The driver did not recognize the *command* specified in the `mt` call.

`2` There was an error during execution of the `mt` command.

ENVIRONMENT VARIABLES

`TAPE` This variable can be set to the name of the special file which the `mt` command is to access if the `–f`

option is not specified.

EXAMPLES

The file `dates/critical` is to be read from the fifth archive on a tape cartridge. The cartridge is in the drive and is currently positioned at the start of the tape:

a) Go to the start of the fifth archive:

```
$ mt –f /dev/ios0/rstape025n fsf 4
```

b) Read the file you want:

```
$ tar xvf /dev/ios0/rstape025 dates/critical
x dates/critical
```

When `tar` finishes executing, the tape is automatically wound back to the start, since the special file `/dev/ios0/rstape025` was specified.

The remaining available capacity of the cassette is to be established for an MC10 drive:

```
$ mt -f /dev/ios0/rstape003 capa
/dev/ios0/rstape003: 2.35 GB of 2.35 GB (100 %) free
```

The `/home` directory is then saved to the tape:

```
$ tar cvf /dev/ios0/rstape003n /home
$ mt -f /dev/ios0/rstape003n capa
/dev/ios0/rstape003: 1.75 of 2.35 GB (75 %) free
```

NOTES

When working with `mt` *and* `cpio`, it is important to ensure that the tape is positioned behind an EOF (end of file) mark after `mt`, before proceeding to work with `cpio`. This behavior complies with MIPS ABI.

FILES

`/dev/rmt/ctape*`

`/dev/rmt/vtape*`

`/dev/ios?/[r]stape*`

SEE ALSO

`cp(1)`, `cpio(1)`, `dd(1)`, `env(1)`, `set(1)`, `tar(1)`, `ioctl(2)`, `write(2)`, `mtio(7)`, `stape(7)`.

User's Reference Manual

mv(1)

NAME

mv - move or rename files

SYNOPSIS

Format 1

mv [*option ...*] *file filenew*

Format 2

mv [*option ...*] *file ...dir*

Format 3

mv [—] *dir dirnew*

DESCRIPTION

mv is used to change the name of a file, or to move a file from one directory to another within the file tree. To be able to use mv you have to have write permission for the directory in which the file resides or the directory to which you want to move it, as appropriate.

mv does not make copies of files that are moved or renamed within the same file system. Instead, it simply modifies the appropriate entries in the parent directory, retaining any links to other files.

OPTIONS

Format 1: Renaming a file

mv [*option ...*] *file filenew*

The options –f and –i override each other. mv is determined by the last specified option.

No option specified:

If you specify an existing file for *newfile* and do not have write permission for it, mv displays the mode of *newfile* and prompts you to confirm whether it should proceed. It only does so if your answer begins with y.

Caution:

If the standard input is not a terminal, no prompt for confirmation is displayed and *newfile* will not be overwritten).

–f If a file named *newfile* already exists, mv will overwrite the existing file even if you do not have write permission for it.

–i (interactive) If you specify an existing file for *newfile*, you will always be asked to confirm whether mv should really proceed.

When the Veritas File System (VxFS) is installed on the system, the following additional option is available:

–e *extent_op*

Specify how to process extent attribute information. This option specifies the required persistence of extent attributes when processing files which have preallocated space or fixed extent sizes. Valid values for *extent_op* are:

warn Issue a warning message if extent attribute information cannot be kept (the default behavior if this option is not specified).

force

Fail the move if extent attribute information cannot be kept.

ignore

Ignore extent attribute information.

–– If the name of a file to be renamed begins with a dash (–), the end of the command-line options must

be marked with --.

file Name of the file to be renamed.

newfile

New name for the file; must not be the same as *file*. If a file named *newfile* already exists, it is overwritten by the contents of *file* if you have write permission for *newfile* (also see -i).

If you specify an existing file for *newfile* and do not have write permission for it, mv will display the mode of *newfile* and prompt you to confirm whether it should proceed. The existing *newfile* is not overwritten unless you answer y. If the -f option is specified, confirmation will not be requested and *newfile* will be overwritten.

If the standard input is not a terminal, confirmation will not be requested and *newfile* will not be overwritten

If the parent directory of *newfile* is writable but has the sticky bit (the t bit) set, one or more of the following conditions must be satisfied:

- the user must own the file
- the user must own the directory
- the user must have write permission for the file
- the user must be a privileged user

Format 2: Moving files and directories to another directory

mv [*options* ...] *file* ... *dir*

The options -f und -i override each other. mv is determined by the last specified option.

-f If a file with the name *file* already exists in *dir*, mv always overwrites the file, whether you have write permission for the existing file or not.

-i (i – interactive) If a file with the name *file* already exists in *dir*, you will be asked whether you really want to execute mv.

-- If *file* begins with a dash, the end of the command-line options must be marked with --.

file Names of files or directories to be moved to directory *dir*. If you name a directory as source, all the files and directories under it are moved recursively.

dir Name of the directory to which the files and/or directories are to be moved. You need write permission for this target directory.

If *dir* is writable but has the t bit (sticky bit) set, one or more of the following conditions must be satisfied:

- the user must own the file
- the user must own the target directory
- the user must have write permission for the file
- the user must be a privileged user

Format 3: Renaming a directory

mv [--] *dir dirnew*

-- If *dir* begins with a dash, the end of the command-line options must be marked with --.

dir Name of the directory to be renamed.

newdir

New name for the directory.

dir and *newdir* do not necessarily belong to the same physical file system; however, they do not have to share the same parent directory.

If a directory named *newdir* already exists, the directory named *dir* is moved to the *newdir* directory

(corresponding to **Format 2**).

ERROR MESSAGES

`mv: cannot rename` *file*
`mv: permission denied`

You do not have write permission for the directory to which *file* is to be moved.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

Answers to yes/no queries must be given in the language appropriate to the current locale.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The file `songs` in the current working directory is to be renamed `popsongs` and moved to the directory `/home/joanne/arts/music`:

```
$ mv songs /home/joanne/arts/music/popsongs
```

Example 2

The files `daisy`, `rose` and `violet` in the current working directory are to retain their names and be moved to the directory `/home/joanne/flowers`:

```
$ mv daisy rose violet /home/joanne/flowers
```

SEE ALSO

`chmod(1)`, `cp(1)`, `cpio(1)`, `find(1)`, `ln(1)`, `rm(1)`.

newform(1) - pwd(1)

newform(1)

NAME

`newform -` change the format of a text file (new format)

SYNOPSIS

`newform` [*option* ...] [*file* ...]

DESCRIPTION

The `newform` command can be used to reformat text files or input lines that are read in from the standard input.

`newform` reformats input text lines in accordance with the specified command-line options and then reproduces them on the standard output.

OPTIONS

No option specified:

`newform` copies all input lines to the standard output unmodified.

Options must be entered individually, separated from one another by blanks (see **EXAMPLES**). Command-line options are sequentially processed in the order specified. They may appear in any order, but the sequence in which they are entered is significant. This means that option sequences such as `–e15 -160`

and –160 -e15 will produce different results.

Except for –s, options may be repeated as often as required.

Options may also be intermingled with optional file names. The effect is the same as when all options precede file names.

Options are applied to all files on the command line.

Expanding tabs to spaces

–i[*tabspec*]

Input tab specification. This option can be used to expand tab characters to spaces, according to the list of tab stops specified in *tabspec*. newform then converts each tab in the input line to the appropriate number of spaces so that the next character is located at a tab stop (see *Example 1*).

tabspec not specified:

The default value for *tabspec* is -8, which means that newform assumes tab stops in columns 1, 9, 17, 25, etc.

tabspec specified:

tabspec may be specified as any of the following values:

–*n*, –0;

n1,*n2*,...;

–a, –a2, –c, –c2, –c3, –f, –p, –s, –u;

––, ––*name*.

–*n* newform assumes tab stops in columns 1, 1+*n*, 1+2**n*, 1+3**n* etc.

n is an integer greater than or equal to 1.

–0 newform expects no tabs; however, if any are found in the input line, they are converted to one space each. The argument –i–0 thus yields the same result as –i–1.

n1,*n2*,...

Tabs are assumed in columns *n1*, *n2*, etc.

n1, *n2*, etc. are integers greater than or equal to 1. If you enter a plus sign (+) before one of the numbers, the value that follows it is added to the last specified number, and the total is used as the new tab stop. The numbers must be entered in ascending order and be separated by commas. One or more numbers (to a maximum of 40) may be specified.

–a Equivalent to the tab stop list 1,10,16,36,72

(for Assembler; applicable on some mainframes)

–a2 Equivalent to the tab stop list 1,10,16,40,72

(for Assembler; applicable on some mainframes)

–c Equivalent to the tab stop list 1,8,12,16,20,55

(for COBOL; normal format)

–c2 Equivalent to the tab stop list 1,6,10,14,49

(for COBOL; compact format)

–c3 Equivalent to the tab stop list

1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67

(for COBOL; compact format with more tab stops than with –c2)

–f Equivalent to the tab stop list 1,7,11,15,19,23

(for FORTRAN)

–p Equivalent to the tab stop list

1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61

(for PL/1)

−s Equivalent to the tab stop list 1,10,55

(for SNOBOL)

−u Equivalent to the tab stop list 1,12,20,44

(for Assembler; applicable on some mainframes)

−− `newform` assumes that the tab specification is to be found in the first line read from the standard input. The tab specification must have the following format:

`<:t`*tabspec* `d:>`

−−*name*

The tab specification is looked for in the first line of the file called *name*. It must have the following format:

`<:t`*tabspec* `d:>`

Converting spaces to tabs

−o[*tabspec*]

Output tab specification. This option can be used to convert spaces to tabs, according to the tab stop list specified in *tabspec*. If one or more spaces precede these tab stops, `newform` converts these spaces to one tab character (see *Example 2*).

tabspec not specified:

The default value for *tabspec* is −8, which means that `newform` assumes tab stops in columns 1, 9, 17, 25, etc.

tabspec specified:

The tab specifications are the same as for `−i`. If you enter a value of −0 for *tabspec*, spaces will not be converted to tabs.

Print tab specification

−f (format line) `newform` writes the tab specification format line on the standard output before displaying any other lines.

The tab specification has the following format:

`<:t`*tabspec* `d:>`

The value printed for *tabspec* will correspond to the format specified in the last `−o` option. If you have not specified a value with the `−o` option or have omitted it entirely, the default value of −8 is printed for *tabspec*.

Caution:

If option `−f` is used, and the last `−o` option specified was `−o−−`, preceded by either `−o−−` or `−i−−`, the tab specification format line will be incorrect.

Example: `newform ... −i−− ... −o−− −f file`

Placing the first field at the end of a line

−s (shear off) `newform` shears off all characters in the input line up to and including the first tab.

If only 8 characters or less (excluding the first tab) are sheared off, all these characters (with the exception of the first tab) are placed at the end of the line.

If more than 8 characters (excluding the first tab) are sheared off, the eighth character is replaced by an asterisk; the following characters are deleted. The first tab is always deleted.

Interaction with other options:

When the `−s` option is processed, `newform` shears off the characters at the start of the input line and saves them internally until all other options specified have been applied to that line. Only then are the

characters added to the end of the processed line.

Caution:

When you specify the –s option, newform expects all input lines to contain a tab character.

Truncating and extending lines

–l[*n*]

The effective line length is set to *n* characters, where *n* is an integer greater than or equal to 1.

The –l option is only meaningful in combination with options –b, –e, –p, or –a.

n not specified:

The effective line length is set to 72 characters.

–l[*n*] not specified:

The effective line length defaults to 80 characters.

Caution:

Tabs and backspaces are considered to be one character each; however, the –i option can be used to expand tabs to a specified number of spaces.

–b[*n*]

(beginning) If the length of the input line exceeds that of the effective line length (see option –l), newform truncates the first *n* characters from the beginning of the line.

n is an integer greater than or equal to 1.

n not specified:

The number of characters needed to obtain the effective line length are truncated from the beginning of the line.

–e[*n*]

(end) If the length of the input line is greater than the effective line length (see option –l), newform truncates the last *n* characters from the end of the line.

n is an integer greater than or equal to 1.

n not specified:

The number of lines needed to obtain the effective line length are truncated from the end of the line.

–c*k* The character used for padding is changed to *k*, where *k* is any printable ASCII character.

The –c option is only meaningful in combination with the –p or –a options.

–c not specified:

The default character for *k* is a space.

–p[*n*]

(prefix) If the length of the input line is less than the effective line length (see option –l), newform prefixes *n* padding characters (see option –c) to the beginning of the line.

n is an integer greater than or equal to 1.

n not specified:

newform prefixes the number of characters required to obtain the effective line length.

–a[*n*]

(append) If the length of the input line is less than the effective line length (see option –l), newform appends *n* padding characters (see option –c) to the end of the line.

n is an integer greater than or equal to 1.

n not specified:

The number of characters required to obtain the effective line length is appended to the end of the line by default.

file Name of the text file to be reformatted.

If you specify more than one file, the options will be applied to all files on the command line.

file not specified:

`newform` reads from the standard input.

ERROR MESSAGES

All the errors described below cause the program to exit.

`not –s format`

You have called `newform` with the `–s` option, but one of the input lines does not contain a tab character.

`can't open` *filename*

The input file *filename* cannot be opened, possibly because it does not exist or because you have no permission to access it, etc.

`internal line too long`

A line exceeds 512 characters when expanded in the internal work buffer.

`tabspec in error`

The *tabspec* argument specified with the `–i` or `–o` option does not have the correct format.

`tabspec indirection illegal`

You have called `newform` with the `–i––`, `–o––`, `–i––`*file*, or `–o––`*file* option; however, the tab specification that was subsequently read by `newform` from the standard input or from *file* contained another *tabspec* in the form `–i––`, `–o––`, `–i––`*file*, or `–o––`*file*. This is not permitted. In other words, a *tabspec* read from a file (or standard input) may not contain a *tabspec* referencing another file (or standard input).

EXAMPLES

Converting tabs to spaces: option -i

Example 1

The contents of `file` are:

`red` <TAB> `green`

There is only one tab between the words `red` and `green`. The command `od` can now be used to check the file contents; `od` displays tabs and newline characters as well:

```
$ od -c file
0000000   r e d  \t  g r e e n  \n
0000012
```

0000012 indicates the number of characters contained in `file` in octal notation.

Compare the output of `cat` and `newform -i`:

```
$ cat file
red     green
$ newform -i file
red     green
```

The word `green` begins in column 9 in both cases. Although it appears that `newform -i` has not changed the contents of the file, the command `od` shows that the tab character has in fact been converted to 5 spaces:

```
$ newform -i file | od -c
0000000   r e d          g r e e n  \n
0000016
```

Now specify a *tabspec* argument with the `–i` option. Note that there should be no space between `–i` and the value for *tabspec*.

```
$ newform -i-4 file
red green
```

Since `newform` assumes the presence of a tab stop in column 5 in this case, the word `green` begins on column 5. The tab character is converted to just one space.

Define a list of tab stops. To do this, you first create a file containing several tabs in one line:

```
$ cat > file1
red <TAB> green <TAB> blue <TAB> yellow<CTRL> <D>$ newform -i5,13,18 file1
red green   blue yellow
```

Here `newform` assumes tab stops in columns 5, 13, and 18. The word `green` thus begins on column 5, i.e. the first tab is converted to one space; the word "blue" begins on column 13, i.e. the second tab is converted to 3 spaces, and so on.

Converting spaces to tabs: option -o

Example 2

The contents of `file` are:

red green

There are 5 spaces between the words `red` and `green`. Convert the spaces to a tab:

```
$ newform -o file | od -c
0000000   r e d  \t   g r e e n  \n
0000012
```

`newform` assumes a tab stop in column 9 and converts all spaces that immediately precede column 9 to 1 tab character.

Now enter a value of -6 for *tabspec*:

```
$ newform -o-6 file | od -c
0000000   r e d  \t   g r e e n  \n
0000014
```

All spaces immediately preceding column 7 have been converted to 1 tab character. This tab is followed by the remaining two spaces.

Truncating and extending lines

Example 3

The contents of `file` are:

monday
tuesday
wednesday
thursday
friday
saturday
sunday

All lines from `file` that are longer than 6 characters are to be truncated to the right so that the effective line length equals 6 characters:

```
$ newform -l6 -e file
monday
tuesda
wednes
thursd
friday
saturd
sunday
```

All lines from `file` are to be output with right justification, ending at column 20:

```
$ newform -l20 -p file
       monday
      tuesday
    wednesday
     thursday
       friday
```

```
     saturday
       sunday
```

The following `newform` call deletes the sequence number area (columns 1-6) from a COBOL program:

```
$ newform -l1 -b6 program
   .
   .
   .
```

The `–l1` option is required to ensure that *every* line is truncated by 6 characters (see *Example 5*).

Rearrange line segments: option -s

Example 4

The contents of `file` are:

```
Mason <TAB> Edward
Montana <TAB> Dave
$ cat file
Mason   Edward
Montana Dave
```

You now want to rearrange the lines so that the last name follows the first name with one intervening space.

```
$ newform -a1 -s file
Edward Mason
Dave Montana
```

First, the `–a1` option appends one space to the end of the lines. The `–s` option then shears off the surname and the tab character, and appends the surname to the end of the line.

The same effect could also have been achieved by specifying `newform -s -a1` *file*. `newform` would have then sheared off the surnames and tabs first, saved the surnames internally, appended a space to the end of the line, and finally added the surnames as well (see *Example 6*).

Sequence of options/repeating options

Example 5

The contents of `colors` are:

```
red
violet
blue
magenta
```

Truncate two characters from the end of all lines that are longer than 4 characters:

```
$ newform -l4 -e2 colors
red
viol
blue
magen
```

A different result is obtained if you switch the two options:

```
$ newform -e2 -l4 colors
red
violet
blue
magenta
```

In this case, the effective line length is first set to the default value of 80, i.e. only lines that exceed 80 characters in length are truncated. There are no such lines in the example. The effective line length is then set to a value of 4; however, this is no longer significant.

Now all the lines in `colors` which have no more than 5 characters are to be prefixed with an asterisk. Then all lines that are longer than 6 characters are to have the last character cut off.

```
$ newform -l5 -c* -p1 -l6 -e1 colors
*red
violet
*blue
magent
```

Example 6

The contents of `orders` are:

```
1000 <TAB> Meyer <TAB> Frankfurt
 150 <TAB> Miller <TAB> London
 750 <TAB> Patel <TAB> Bombay
2500 <TAB> Smith <TAB> Perth <TAB> ***
```

The lines in the file are now to be reformatted as follows:

The numbers and initial tabs are to be truncated so that the names come first. The remaining tabs are to be expanded into spaces, with tab stops set every 15 columns (i.e. in columns 16, 31, 46, etc.). All lines are then to be padded up to column 30 with spaces, or truncated from column 31 onward as required. The previously sheared off numbers are then to be appended to each line starting at column 31.

This can be accomplished by calling `newform` as follows:

```
$ newform -s -i-15 -l30 -a -e orders
Meyer          Frankfurt      1000
Miller         London          150
Patel          Bombay          750
Smith          Perth          2500
```

Note that the options are processed from left to right. When the numbers and tabs have been sheared off (option `-s`), the names begin on column 1. Following the expansion of the remaining tabs to spaces (option `-i-15`), only the first three lines are shorter than 30 characters and hence padded with spaces (option `-a`); the last line exceeds 30 characters in length and is therefore truncated at the end (option `-e`). The initially sheared off numbers are finally appended to the end of each relevant line (option `-s`, step 2).

SEE ALSO

`csplit`(1), `tabs`(1), `fspec`(4).

User's Reference Manual

newgrp(1)

NAME

`newgrp` - change to new group

SYNOPSIS

`newgrp` [–] [*group*]

DESCRIPTION

The shell built-in `newgrp` overlays the current shell with `/bin/newgrp`. The `/bin/newgrp` command makes the specified group ID number your current group ID and overlays itself with a new shell. You terminate the shell in which you called `newgrp` by hitting the CTRL-D key.

Note:

If `newgrp` is killed with the DEL key while `/bin/newgrp` is replacing the current shell, the shell from which `newgrp` was called is terminated as well.

The `newgrp` command can thus be used to switch to another user group. This means:

- your access permissions for existing files will be changed to those of the new group affiliation;
- group access permissions for new files that you create will be those of the group to which you have switched.

Only the variables that you exported earlier [see `export(1)`] will be known in the (new) current shell after the group has been changed. Variables that have not been exported are either considered undefined or assigned a default value by the shell [see `sh(1)`]. Shell variables such as `PATH` and `HOME` are also assigned default values, unless they have already been exported by the system or were explicitly exported by you. Also, if user's real user id is not 0, i.e. not `root`, the file size limit is set to `ULIMIT` in `/etc/default/login` or system default value if there is no `ULIMIT`.

Before the call

The `/etc/group` file must contain an entry for the group that you wish to switch to. Otherwise, `newgrp` will terminate with an error message.

The `newgrp` command can be used to switch to any group to which you belong, i.e. any group for which your login name appears in the `/etc/group` file under the corresponding group entry. If a password has been defined for this group in the `/etc/group` file, `newgrp` expects you to enter this password before making the switch.

If you are not a member of a group, you can only switch to it if it has a password. Otherwise, `newgrp` exits with an error message.

OPERANDS

No argument specified:

Changes you back to the group whose group ID (GID) has been entered for your login name in the `/etc/passwd` file.

– You may only use this argument if a shell is started for your login name at login time.

The `newgrp` command overlays the current shell with a login shell. Before this shell displays its prompt, it first executes the `/etc/profile` and your `$HOME/.profile` (if present) and switches to your HOME directory.

In other words, except for the fact that you are now a member of a new group (the one specified on invoking `newgrp`), you continue working in the same environment as the one that applied after you logged into the system.

– not specified:

`newgrp` overlays the current shell with `/bin/newgrp`. The current directory is not changed; however,

the new shell will not be aware of variables that have not been explicitly exported. Unexported variables are either undefined or assigned a default value by the shell.

group

Name of the group you wish to switch to. There must be an entry for this group name in the `/etc/group` file. The associated group ID (GID) must already be associated with a login name in the `/etc/passwd` file.

If you are not a member of the specified group, there must be a password defined for the group in the `/etc/group` file. `newgrp` expects you to enter this password before it switches groups.

If there is no password for your login name in the `/etc/passwd` file, it is essential that a password exists for the group in the file `/etc/group`. The `newgrp` command expects you to enter this password before it switches groups.

To switch back to the user group that is entered for you in the `/etc/passwd` file, simply call `newgrp` without a group name.

group not specified:

You are returned to the group whose group ID (GID) is entered in the `/etc/passwd` file for your login name.

Setting up a group password

If you want to switch to a group of which you are not a member, the system administrator will have to set up a password for that group.

For the system administrator only

Since no command is available to define a password for a group, you will have to resort to a few "tricks". Two options are available:

- You can simply copy an encrypted password from the `/etc/passwd` file, provided you know its meaning in plain text.
- You can install a "dummy password" in the `/etc/passwd` file and use the `passwd` command to define a password. You then write this encrypted password into the `/etc/group` file.

EXIT STATUS

As soon as `newgrp` generates a new shell environment, its exit status is the same as the shell's, regardless of whether it is possible to switch to the new group or not.

`>0` An error occurred.

ERROR MESSAGES

`Unknown group`

This name has not been entered in the `/etc/group` file.

`Sorry`

You are not permitted to switch to this group, since you are not a member of the group, and no password has been defined for it either.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Change to the group with the group name `council`:

```
$ newgrp council
```

```
$ >newfile
$ chmod 640 newfile
$ ls –lg newfile
-rw-r----- 1 rose   council      162 Mar 19 18:34 newfile
```

The new file (`newfile`) created after the group change is available to members of the `council` group as a read-only file.

NOTES

Some differences in behavior may occur when using `newgrp`, depending on which shell is being used. The possible differences are not described specifically.

FILES

`/etc/default/login`
: Contains the default settings for the `login(1)` command.

`/etc/default/passwd`
: Contains the default settings for the `passwd(1)` command.

`/etc/group`
: Defines a group name for group IDs entered in the `/etc/passwd` file as well as all members allowed in the group.

`/etc/passwd`
: Contains all installed login names and group IDs.

SEE ALSO

`exec(1)`, `export(1)`, `ksh(1)`, `login(1)`, `sh(1)`, `default_passwd(4)`, `group(4)`, `login(4)`, `passwd(4)`, `environ(5)`.

User's Reference Manual

news(1)

NAME

news - output news items

SYNOPSIS

news [*option ...*] [*news_items ...*]

DESCRIPTION

news outputs news items from the system news pool /var/news; the most recent items are output first.

OPTIONS

No option specified:

news outputs the contents of all the files residing in the directory /var/news which are more recent than the time at which news was last called.

The news items are output on a last-in first-out basis (i.e. most recent first).

Each item has an appropriate header.

When news is called for the first time, it creates a file named .news_time in the calling user's home directory (as determined by the $HOME variable). In subsequent calls news uses the time at which this file was last accessed (the "currency" time) to select the items to be displayed to the user. Only those items which are more recent than this access time are output, i.e. only those items more recent than the time news was last invoked.

Each time news is called, the access time of the .news_time file is changed to the current system time.

−a (all) news outputs all news items, regardless of their currency. If this option is set, the access time of the .news_time file is not changed.

−n (names) news only outputs the names of the current news files, not their contents. The access time of the .news_time file is not changed.

−s (statistics) news outputs the number of current news items that exist, without listing their file names or contents, and without changing the access time of the .news_time file.

It is useful to include such an invocation of news in each user's .profile file, or in the system's /etc/profile.

news_items

news_items designates the individual files from the /var/news directory whose contents are to be output. More than one item may be specified.

news_items not specified:

news outputs the contents of files residing in the /var/news directory which are more recent than the time at which news was last invoked.

Functionality

news displays the contents of individual news items one after the other on the screen.

Output of an item can be stopped by hitting <DEL>. news then terminates output of the current item and starts again with the next item. If you press <DEL> twice in quick succession news aborts completely.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

The LC_TIME environment variable governs the format of date and time strings.

If LC_MESSAGES or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If

LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

To check if there are news items in the system news pool, you enter:

```
$ news –s
5 news items
```

Example 2

To list the names under which these items are stored, you enter:

```
$ news –n
news: minutes party safety important ignore
```

To display the file ignore on the screen, you enter:

```
$ news ignore | pg
important (harry) Mon Aug 12 14:24:19 1991
   Curiosity killed the cat!
   Check the file "important" for relevant news!
   Your system administrator
```

FILES

/etc/profile
: File executed by every login shell. This file is created by the system administrator.

/var/news/*
: News files.

$HOME/.news_time
: File with information on the last invocation of news.

SEE ALSO

profile(4), environ(5).

User's Reference Manual

nice(1)

NAME

`nice -` run a command at low priority (be nice)

SYNOPSIS

`nice` [–`n` *increment*] [––] *command*

DESCRIPTION

The priority of a process determines when it is allocated to the next available processor. By default, each process has a priority of 20. `nice` can be used to influence the priority with which a command is executed. High priority values correspond to low priorities and vice versa. Priority values are defined by means of various parameters:

- the time a process has been allocated to the processor
- the time a process has been waiting for the processor
- the value you specify in the `nice` command line.

The invoking process (generally the user's shell) must be in the time-sharing scheduling class [see `priocntl(1)`]. The specified *command* is executed in the time-sharing class.

If you are not a system administrator, you can only use `nice` to increase the priority number, i.e. to decrease the priority of the process (this is known as being "nice" to other users).

A system administrator can also reduce the priority number and thus increase the priority.

OPTIONS

–`n` *increment*

Determines the priority for *command*. *increment* is an integer.

For users without special privileges:

The *increment* argument lies in the range 1-19. The priority number is increased, i.e. the specified command is run with a lower priority. `nice` treats an *increment* greater than 19 as equivalent to 19.

For the system administrator only:

increment lies in the range –20 to 19. Negative *increments* decrease the priority number and hence raise the corresponding priority. `nice` treats an *increment* that is less than –20 as equivalent to –20.

It treats an *increment* greater than 19 as equivalent to 19.

This corresponds to the old option –*increment* which is still supported.

Included in the output of the `ps –l` command is the NI column. This column displays a numeric value between 0 and 39, where:

`0` corresponds to `nice –n –20`

`39` corresponds to `nice –n 19`

–`n` not specified: An *increment* of 10 is assumed by default.

–– If `command` begins with a dash (–), the end of the command line options must be marked with ––.

command

Any command or shell script.

EXIT STATUS

`nice` returns the exit status of the executed *command*. Otherwise it returns one of the following values:

`1-125`

An error occurred with the command `nice`.

`126` The specified *command* exists but cannot be executed.

`127` The specified *command* could not be found.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The command `ps –ale` is to run with lower priority:

```
$ nice –n 15 ps –ale
```

Example 2

The system administrator now runs the same command with increased priority:

```
# nice –n –5 ps –ale
```

SEE ALSO

`ps(1)`, `nohup(1)`, `priocntl(1)`, `renice(1)`, `nice(2)`.

User's Reference Manual

nl(1)

NAME

`nl` - number lines

SYNOPSIS

`nl` [*option ...*] [*file*]

DESCRIPTION

The `nl` command reads lines from a file or standard input and writes them to standard output with line numbering.

`nl` views the text it reads in terms of logical pages. A logical page consists of a header, a body, and a footer section. Empty sections are valid.

The start of a header, body, and footer of a logical page is normally signaled by an input line containing nothing but one of the following strings:

Line	Start of
`\:\:\:`	header
`\:\:`	body
`\:`	footer

If the input text does not contain any delimiter characters, `nl` assumes that the text being read is in a single logical page body.

Line numbering is reset at the start of a logical page (exception: option `–p`). Different line numbering options are independently available for header, body, and footer (e.g. no numbering of header and footer lines, while numbering blank lines only in the body).

OPTIONS

No option specified:

`nl` numbers all logical page body lines that contain printable text, but not header and footer lines.

Line numbering is reset to 1 at the start of each logical page.

Within a logical page, `nl` numbers lines in increments of 1.

Each line number may be up to 6 positions long and is output right justified without leading zeros. A tab is used to separate line numbers from text.

The options must be entered individually, i.e. separated from other options by blanks. The name of the file may be specified before, between, or after the options. The position of the file name in the command line has no effect on the operation of the `nl` command.

Selecting the lines to be numbered

`–b` *type*

Specifies which logical page body lines are to be numbered.

type can be any of: `a`, `n`, `p`*reg_expr*, or `t`.

`a` All lines are numbered.

`n` No lines are numbered.

`p`*reg_expr*

Numbers all lines containing strings that match the given *reg_expr*.

reg_expr is a simple (non-internationalized) regular expression [see `expressions(5)`]. If the specified regular expression contains shell metacharacters, it must be enclosed in single quotes:

p'*reg_expr*'.

`t` All lines with printable text are numbered.

`-b` not specified:

Only the lines with printable text are numbered. Option `-b` with the *t* argument is thus the default.

`-f` *type*

Specifies which logical page footer lines are to be numbered.

type may be: `a`, `n`, p*reg_expr*, or `t`.

Option `-f` not specified:

Logical page footer lines are not numbered. Option `-f` with the *n* argument is thus the default.

`-h` *type*

Specifies which logical page header lines are to be numbered.

type may be: `a`, `n`, p*reg_expr*, or `t`.

`-h` not specified:

Logical page header lines are not numbered. Option `-h` with the *n* argument is thus the default.

Restarting line numbering

`-p` Line numbering is not restarted at logical page delimiters.

`-v` *startnum*

Line numbering is restarted at *startnum* at the start of each logical page.

startnum may be any number greater than or equal to 0.

`-v` not specified:

The line counter is reset to 1 at the start of each logical page.

Defining the increment

`-i` *incr*

incr is the increment value used to number logical page lines.

`-i` not specified:

The default increment is 1.

Defining the output format

`-n` *format*

format is the line numbering format.

The recognized values are: `ln`, `rn`, or `rz`.

`ln` Prints line numbers left justified, suppressing leading zeros.

`rn` Prints line numbers right justified, suppressing leading zeros.

`rz` Prints line numbers right justified, but with leading zeros.

`-n` not specified:

Line numbers are printed right justified, without leading zeros. In other words, the `-n` option with argument `rn` is the default.

`-s` *sep*

Defines *sep* as the separator between line numbers and the corresponding text lines. *sep* may be one or more characters.

`-s` not specified:

By default, line numbers and text lines are separated by a tab character.

`-w` *n* Defines *n* as the number of positions for line numbers.

The maximum value for *n* is 100. If you specify a higher value, a value of 100 is assumed.

`-w` not specified:

Individual line numbers may be up to 6 positions long.

Numbering blank lines

`-l` *n* `nl` treats *n* consecutive blank lines as a single blank line.

Example:

```
$ nl -b a -l 2
```

results in only every second blank line being numbered when there are several consecutive blank lines (none of the header and footer lines are numbered).

`-l` not specified:

Each blank line is interpreted as one full line (*n* = 1).

Defining delimiters for header, body and footer

`-d` *x*[*y*]

Changes the delimiter characters that specify the start of a header, body or footer section from \: to the string *xy*.

If a backslash (\) is to be specified for *x* or *y*, it must be escaped by single quotes or a second backslash, e.g. `-d'\*'` or `-d\\*`.

y not specified:

The start of a header, body or footer section is identified by the string *x*: instead of \:, i.e. the second character retains its default value.

`--` If *file* begins with a dash (-), the end of the command line options must be marked with `--`.

file Name of the input file.

file not specified:

`nl` reads from standard input.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

`LC_COLLATE` and `LC_CTYPE` determine the behavior of ranges, equivalence classes and character sets or character classes within regular expressions.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file `poems` has the following contents:

```
\:\:\:
Taken from: Do you love me?
\:\:
The trouble with you
's you've lost a screw
I'm sorry it's you
but there's nothing to do
There'll be no abatements
there are no replacements
don't make a to-do
just say toodle-oo
I'm sorry I can't help you
you'd cost too much to redo
```

```
you'll have to be abolished
report to be demolished
\:
Ronald Laing
\:\:\:
Limerick
\:\:
A limerick packs laughs anatomical
into space that is short and economical;
But the good ones I've seen
so seldom are clean,
And the clean ones so seldom are comical.
\:
Author unknown
```

Call nl without options

Example 1

```
$ nl poems
Taken from: Do you love me?
     1  The trouble with you
     2  's you've lost a screw
     3  I'm sorry it's you
     4  but there's nothing to do
     5  There'll be no abatements
     6  there are no replacements
     7  don't make a to-do
     8  just say toodle-oo
     9  I'm sorry I can't help you
    10  you'd cost too much to redo
    11  you'll have to be abolished
    12  report to be demolished
Ronald Laing
Limerick
     1  A limerick packs laughs anatomical
     2  into space that is short and economical;
     3  But the good ones I've seen
     4  so seldom are clean,
     5  And the clean ones so seldom are comical.
Author unknown
```

Number lines in increments of 10

Example 2

```
$ nl –v 10 –i 10 poems
Taken from: Do you love me?
    10  The trouble with you
    20  's you've lost a screw
    30  I'm sorry it's you
    40  but there's nothing to do
    50  There'll be no abatements
    60  there are no replacements
    70  don't make a to-do
    80  just say toodle-oo
    90  I'm sorry I can't help you
   100  you'd cost too much to redo
   110  you'll have to be abolished
   120  report to be demolished
```

```
Ronald Laing
Limerick
    10  A limerick packs laughs anatomical
    20  into space that is short and economical;
    30  But the good ones I've seen
    40  so seldom are clean,
    50  And the clean ones so seldom are comical.
Author unknown
```

SEE ALSO

ed(1), pr(1), expressions(5).

User's Reference Manual

nohup(1)

NAME

nohup - run a command immune to hangups and quits

SYNOPSIS

nohup *command* [*argument* ...]

DESCRIPTION

nohup executes a command or shell script, ignoring the signals SIGHUP (signal number 1) and SIGQUIT (signal number 3). nohup can be used to prevent a process from being terminated when you log out from a terminal.

Before the call

If you wish to use nohup with a pipe or a list of commands, you must put the relevant pipeline or command list in a file and run the file as a shell script under nohup's management.

ARGUMENTS

command [*argument* ...]

Any command or shell script. You can pass arguments to *command* by entering them after the command as usual.

If the standard output and standard error of *command* have not been redirected, they will be written to a file called nohup.out in the current directory.

If you are not permitted to create the file nohup.out in the current directory (i.e. you have no write permission for this directory) or if you do not have write permission for the file nohup.out in the current directory, the output will be redirected to the file $HOME/nohup.out\.

If the file nohup.out or $HOME/nohup.out does not exist, a corresponding file is created which has exclusive permissions for the user executing the call; otherwise, the output is appended to the existing file.

EXIT STATUS

126 The specified *command* exists, but cannot be executed.

127 The specified *command* cannot be found or an error occurred for the nohup command.

If *command* was called, nohup returns the exit status of the executed command *command*.

ERROR MESSAGE

```
nohup: cannot open/create nohup.out
```

You are not allowed to create a file called nohup.out in the current working directory or in $HOME, or you are not allowed to write to existing nohup.out files in those directories.

ENVIRONMENT VARIABLE

HOME The value of the environment variable HOME is used to determine the directory in which the file nohup.out is to be created in cases where nohup.out in the current directory is write-protected or where you are not allowed to create files in the current directory.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLE

You initiate the following job and set it to run in the background:

```
$ nohup program &
sending output to nohup.out
```

You then log out. `program` is not terminated, since it is immune to the quit signal. The output is written to a file called `nohup.out`.

FILES

`nohup.out`
: File located in the current directory which receives the standard output and standard error output from commands executed under `nohup`.

`$HOME/nohup.out`
: The standard output and standard error output of commands executed under `nohup` are sent to this file if the `nohup.out` file in the current directory is write-protected or if you are not allowed to create this file in the current directory (because you do not have write permission there).

SEE ALSO

`chmod`(1), `kill`(1), `nice`(1), `sh`(1), `signal`(2).

User's Reference Manual

notify(1)

NAME

`notify -` notify user of the arrival of new mail

SYNOPSIS

Format 1

`notify –y [ –m` *mailfile* `]`

Format 2

`notify [ –n ]`

DESCRIPTION

`notify` informs the user of the arrival of new mail messages, i.e. of electronic mail sent by the `mail` command.

When a new mail message arrives, `mail` first checks if this message is to be forwarded to another recipient or is to be used as the input for a command. `notify` is used to set up forwarding to an alternative mailbox.

OPTIONS

No option specified:

`notify` reports on the status of the automatic mail notification facility, indicating whether or not it is activated.

`–m` *mailfile*

(m – mailfile) This option can be used to specify a file in which mail messages are to be saved while automatic notification is activated.

mailfile not specified:

The default file for saving messages is `$HOME/.mailfile`.

`–n` Removes the mail notification facility.

`–y` Installs the mail notification facility.

Mode of operation

`notify` first checks the system file `/var/adm/utmp` to determine if the recipient is currently logged in. If so, the terminal on which the recipient is working (this is also determined from the entry in `/var/adm/utmp`) is opened for writing. The user is then notified about the new message. The notification will indicate who the message is from. If the message contains a "Subject:" header line, the subject will also be included (for security reasons, all unprintable characters in the header will be converted to exclamation points).

If the user is logged in more than once, he or she will get multiple notifications, one per terminal. The `mesg -n` command can be used to disable notifications for a particular login session. This command removes the write permission to the terminal specified.

If more than one system is connected via NFS (Network File System), `notify` checks the `/var/adm/utmp` files on all the systems connected.

If problems are encountered when delivering mail to the appropriate mailbox file, `notify` checks the directory of the mailbox file in the system file `/etc/mail/notify.fsys`. If the file's directory is found in the first column of the file, the message is forwarded to the system listed in the second column of the file. If it is not found here, the message is returned to the sender.

EXAMPLES

Example 1

To check if `notify` is activated:

```
$ notify
```

```
/usr/bin/notify: Mail notification not active
```

Example 2

To activate the mail notification facility:

```
$ notify -y
/usr/bin/notify: Asynchronous 'new mail' notification installed
/usr/bin/notify: New mail messages will go to '/$HOME/.mailfile'
/usr/bin/notify: Be sure to re-define ${MAIL} appropriately
```

FILES

`/tmp/notif*`
: Temporary file.

`/var/mail/*`
: Users' standard mailboxes.

`/usr/lib/mail/notify2`
: Program that performs the notification.

`/etc/mail/notify.fsys`
: List of file systems.

`/var/adm/utmp`
: List of users who are logged in.

SEE ALSO

`mail(1)`, `mesg(1)`.

User's Reference Manual

od(1)

NAME

`od -` dump contents of a file in various formats

SYNOPSIS

Format 1

`od` [`–v`] [`–A` *format*] [`–j` *number*] [`–N` *number*] [`–t` *format ...*] [`––`] [*file ...*]

Format 2

`od` [*option ...*] [*file*] [[`+`]*offset*[`.`][`b`]]

DESCRIPTION

The `od` command dumps (i.e. writes the contents of) a file on the standard output in an output format that you select with command-line options.

The first column of each output line specifies the position of the first character in this line. Characters can be specified in octal, decimal or hexadecimal, depending on the indicated output format.

OPTIONS

Format 1: New synopsis for od

No option specified:

Each group of 2 bytes is interpreted as an unsigned octal number (same as option `–t`□`o2`).

`–A` *format*

Defines the format for the positioning specification in the first column of each line of output. *format* can assume the following values:

`d` Decimal positioning specification

`o` Octal positioning specification

`x` Hexadecimal positioning specification

`n` No positioning specification

`–j` *number*

Defines the position in the file at which output is to begin. *number* specifies how many bytes are to be skipped at the beginning. Multiple files are handled as if joined together in a single file.

number is interpreted as a decimal number by default. This format can be varied by means of additional specifications:

`0x`*number* or `0X`*number*

number is interpreted as a hexadecimal number

`0`*number*

number is interpreted as an octal number

number`b`

number is interpreted as a multiple of 512 bytes (cannot be combined with `0x`, as `b` is interpreted as part of the hexadecimal number in this case)

number`k`

number is interpreted as a multiple of 1024 bytes

number`m`

number is interpreted as a multiple of 1024x1024 bytes

`–N` *number*

`od` outputs a maximum of *number* bytes.

number is interpreted as a decimal number by default. This format can be varied by specifying `0x`, `0X`, or `0` before *number*:

`0x`*number* or `0X`*number*
: *number* is interpreted as a hexadecimal number

`0`*number*
: *number* is interpreted as an octal number

`–t` *format*
: Defines the output format. You can specify a number of output formats or options `–t`. The lines are output in the sequence of the specified output formats or `–t` options.

format can assume the following values:

`a`
: Output as characters; names in accordance with ISO/IEC 646 are output instead of control characters [see `ascii(5)`]:

Oct.	Name	Meaning
\000	`nul`	Null
\001	`soh`	Start of Heading
\002	`stx`	Start of Text
\003	`etx`	End of Text
\004	`eot`	End of Transmission
\005	`enq`	Enquiry
\006	`ack`	Acknowledge
\007	`bel`	Bell
\010	`bs`	Backspace
\011	`ht`	Horizontal Tabulation
\012	`lf` or `nl`	Line Feed
\013	`vt`	Vertical Tabulation
\014	`ff`	Form Feed
\015	`cr`	Carriage Return
\016	`so`	Shift Out
\017	`si`	Shift In
\020	`dle`	Data Link Escape

Oct.	Name	Meaning
\021	`dc1`	Device Control 1
\022	`dc2`	Device Control 2
\023	`dc3`	Device Control 3
\024	`dc4`	Device Control 4
\025	`nak`	Negative Acknowledge
\026	`syn`	Synchronous Idle
\027	`etb`	End of Transmission Block

\030	can	Cancel
\031	em	End of Medium
\032	sub	Substitute Character
\033	esc	Escape
\034	fs	File Separator
\035	gs	Group Separator
\036	rs	Record Separator
\037	us	Unit Separator
\040	sp	Space
\177	del	Delete

`c` Output as characters; some control characters are represented as escape sequences:

Null byte (all bits are 0)	`\0`
Warning, alert	`\a`
Backspace	`\b`
Form feed	`\f`
Newline character	`\n`
Carriage return	`\r`
Horizontal tabs	`\t`
Vertical tabs	`\v`

The remaining non-printing characters are output as three-digit octal numbers [see `meta-ascii(5)`].

The number of bytes that are interpreted as a single character depends on the internationalized environment.

`d` Signed decimal number

`f` Floating point number

`o` Octal number

`u` Unsigned decimal number

`x` Hexadecimal number

For the formats `d`, `f`, `o`, `u`, and `x` you can define how many bytes are to be considered as a single unit. To do this you specify a number after the format letter, e.g. `-t o2`.

The floating point number type (`f`) is defined by entering one of the following letters:

`F` Floating point number of type `float`

`D` Floating point number of type `double`

`L` Floating point number of type `long double`

The other numeric representation types (`d`, `o`, `u`, `x`) are defined by specifying one of the following letters:

`C` Number of type `char`

`S` Number of type `short`

`I` Number of type `int`

`L` Number of type `long`

–t not specified:

Each group of 2 bytes is interpreted as an unsigned octal number (same as option –t o2).

–v (v – verbose) All data is displayed.

–v not specified:

Any number of lines that are identical with the previous line are replaced by an asterisk (*).

–– End of the list of options. Must be specified if *file* begins with –.

file Name of the file to be output. Several files can be specified.

file not specified: od reads from the standard input.

Warning: If no options are specified and *file* begins with a digit or +, it is interpreted as *offset* in accordance with the old synopsis for od.

Format 2: Old synopsis for od

In previous versions of od, only this format was available. It is still supported, but should not be used by portable applications.

If you specify several options in order to combine various output formats, the dash – must only be specified once, followed by the option names without blanks, e.g. od –bcs *file*. The lines are output in accordance with the sequence of options.

No option specified

Each group of 2 bytes is interpreted as an unsigned octal number (same as option –o).

–b Interprets each byte as an octal number. (Corresponds to the –t o1 option in the new synopsis).

–c Bytes are interpreted as characters. The number of bytes that are interpreted as a single character depends on the internationalized environment. Certain non-printing characters appear as the C escapes shown below:

Null byte (all bits are 0)	\0
Warning, alert	\a
Backspace	\b
Form feed	\f
Newline	\n
Carriage return	\r
Horizontal tabs	\t
Vertical tabs	\v

The other non-graphic characters appear as 3-digit octal numbers [see meta-ascii(5)]. (Corresponds to the –t c option in the new synopsis).

–d Interprets each group of 2 bytes as an unsigned decimal number. (Corresponds to the –t u2 option in the new synopsis).

–D Interprets each group of 4 bytes as an unsigned decimal number. (Corresponds to the –t u4 option in the new synopsis).

–f Interprets each group of 4 bytes as a floating point number. (Corresponds to the –t f4 option in the new synopsis).

–F Interprets each group of 8 bytes as an extended precision number. (Corresponds to the –t f8 option in the new synopsis).

–o Interprets each group of 2 bytes as an unsigned octal number. (Corresponds to the –t o2 option in the new synopsis).

−O Interprets each group of 4 bytes as an unsigned octal number. (Corresponds to the −t□o4 option in the new synopsis).

−s Interprets each group of 2 bytes as an signed decimal number. (Corresponds to the −t□d2 option in the new synopsis).

−S Interprets each group of 4 bytes as an signed decimal number. (Corresponds to the −t□d4 option in the new synopsis).

−v (verbose) Shows all data.

−v not specified:

Any number lines that are identical to the previous line are replaced by an asterisk (*).

−x Interprets each group of 2 bytes as an unsigned hexadecimal number. (Corresponds to the −t x2 option in the new synopsis).

−X Interprets each group of 4 bytes as an unsigned hexadecimal number. (Corresponds to the −t x4 option in the new synopsis).

−− If *file* begins with a dash (−), the end of the command line options must be marked with −−.

file Name of the file to be dumped.

file not specified: od reads from standard input.

[+]*offset*[.][b]

The *offset* argument specifies the offset in the file where dumping is to commence. (Corresponds to the −j option in the new synopsis).

offset is normally interpreted in octal bytes. If a period (.) is appended to the *offset* argument, the offset is interpreted in decimal. If b is appended, the *offset* is interpreted in blocks of 512 bytes.

If the *file* argument is omitted, the offset argument must be preceded by a plus sign (+) to prevent it from being interpreted as a file name.

offset not specified:

Dumping commences at the start of the file.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If C_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Unsigned octal dump of the contents of file text:

```
$ cat text□
Keep a stiff upper lip.□
$ od text□
0000000 062513 070145 060440 071440 064564 063146 072440 070160□
0000020 071145 066040 070151 005056□
0000030
```

Example 2

Display the contents of the file text in octal form and as ASCII characters from the sixth byte onward:

```
$ od −j 5 −t o1 −t c text□
0000005 141 040 163 164 151 146 146 040 165 160 160 145 162 040 154□
          a       s   t   i   f   f       u   p   p   e   r       l□
0000021 151 160 056 012 000□
```

```
                i    p    .   \n   \0□
0000024
```

SEE ALSO

sed(1), ascii(5), meta-ascii(5).

User's Reference Manual

pack(1)

NAME

`pack` - compress files

SYNOPSIS

`pack` [–`f`] [––] [–] *file...*

DESCRIPTION

`pack` compresses files using Huffman coding.

On systems which evaluate the `RSTCHOWN` variable, the user who invokes `pack` is made the owner of the packed file (unless the invoking user is the system administrator). On other systems the owner stays the same.

The access permissions and the dates of last access and modification stay the same.

Where possible, each file is replaced by a file with the same basename plus a `.z` extension.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Typically, text files are reduced to 60-75% of their original size.

Since a decoding tree forms the first part of a packed file, it is usually not worthwhile to pack small files, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures. Object files, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

No packing will occur if

- the file to be compressed is not a regular file
- there are links to the file to be compressed
- the file to be compressed already has a `.z` extension
- the file to be created already exists
- no storage space is saved as a result of compression.

You can use `unpack` to restore packed files to their original form.

`pcat` writes packed files in their original form to standard output without modifying the packed file.

Note: You can also compress files using `compress` and uncompress then using `uncompress`.

OPTIONS

No option specified:
The files specified are compressed if storage space can be saved as a result. The percentage of compression for each packed file is displayed.

–`f` (f – force) Forces *file* to be compressed even though no storage space will be saved.

–`f` not specified:

Only the files for which storage space can be saved are compressed.

–– End of the list of options. Must be specified if *file* begins with – and the argument – is not specified.

– Causes detailed statistical information on the completed compression to be written to standard output:

`pack: file: xx.xx% Compression`
Percentage of compression

`pack: From xxxxx to yyyyy bytes`
Old size versus new size

`Huffman tree has xx levels below root`
Depth of the tree processed with the Huffman algorithm

xx `distinct bytes in input`
: Number of distinct bytes in *file*

`Dictionary overhead =` *xxx* `bytes`
: Size of decoding rules

`Effective entropy =` *x.xx* `bits/byte`
: Actual reduction factor

`Asymptotic entropy =` *x.xx* `bits/byte`
: Theoretical reduction factor

A line containing the following information is output for each character that appears in the input file:

`Byte` *xxxx*
: Octal value of the character

`Count =` *xx*
: Total number of these characters in the file

`Relative freq. =` *x.xxx*
: Relative proportion of these characters to the file size in percent

`Code =` *xxxxx*
: Specifies how the character was coded

– is not an option, but an argument, and must therefore not be placed before the `-f` option. – can be specified before each file and/or before several files in sequence. Every second occurrence of – switches off the output of statistical information again (see *Example 1*).

– not specified: The percentage of compression is printed.

file Name of the file to be compressed. You may specify more than one file.

file must not be a directory and there must be no other links to it.

The packed file is given the name *file*`.z`; the original *file* is deleted after successful compression. *file*`.z` has the same access permissions and the same access and modification times as *file*. The maximum length of the name of *file* is equal to the maximum file name length supported by the file system, minus 2 characters. This allows for the extension to *file*`.z`. If this length is exceeded, *file* will not be compressed.

file can also be specified as *file*`.z`. The suffix `.z` is ignored by `pack`.

Caution:
: Packed files are not necessarily portable to other systems.

  Make sure that the other system has the same character set loaded and has the `unpack` command available on it.

ERROR MESSAGES

`pack:` *file*`: no saving - ` *file* `unchanged`
: (printed to standard output) No storage space can be saved by compression. Compression does not take place. If you wish, you can force the compression with the `-f` option.

`pack:` *file*`: file name too long`
: The name of the file to be packed is too long. Compression does not take place.

`pack:` *file*`.z: already packed`
: The file specified has already been compressed. It cannot be compressed a second time.

`pack:` *file*`: has links`
: There are links to the files to be compressed. Compression does not take place.

`pack:` *file*`: cannot pack a directory`
: The specified file is a directory. Compression does not take place.

`pack:` *file*`: cannot open`
: You have no read permission for the file to be packed, or *file* does not exist.

`pack: file.z: cannot create`
: The packed file cannot be created, e.g. because you have no write permission for the current directory.

`pack: file: read error`
: An error occurred when reading the original file. Compression does not take place.

`pack: file.z: write error`
: An error occurred when writing the packed file. The compression is aborted, and *file* remains unchanged. It is possible that an incomplete *file*`.z` is now present on the system.

`pack: file: cannot pack a zero length file`
: The file to be packed is empty.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Three files are to be compressed. Statistical information on the compression is to be displayed for the first two files, but not for the third. Since the second and subsequent occurrences of the – option serve as toggles, the option must be specified again before `file3` to turn off the statistics function:

```
$ pack – file1 file2 – file3
pack: file1: 40.6% Compression
pack:   From 31960 to 18984 bytes
        Huffman tree has 15 levels below root
        89 distinct bytes in input
        Dictionary overhead = 111 bytes
        Effective  entropy  = 4.75 bits/byte
        Asymptotic entropy  = 4.72 bits/byte
        Byte 0000, Count = 45211, Relative freq. = 0.311, Code = 11
        Byte 0001, Count = 2530, Relative freq. = 0.017, \
        Code = 100000
        ...
        Byte 0377, Count = 2018, Relative freq. = 0.014, \
        Code = 101001
pack: file2: 41.6% Compression
pack:   From 249569 to 145840 bytes
        Huffman tree has 14 levels below root
        94 distinct bytes in input
        Dictionary overhead = 115 bytes
        Effective  entropy  = 4.67 bits/byte
        Asymptotic entropy  = 4.67 bits/byte
        Byte 0000, Count = 96281, Relative freq. = 0.323, Code = 11
        Byte 0001, Count = 5225, Relative freq. = 0.018, \
        Code = 011100
        ...
        Byte 0377, Count = 4644, Relative freq. = 0.016, \
        Code = 101001
pack: file3: 41.7% Compression
```

Example 2

All files contained in the directory `GMC3` are to be compressed. The file `print` in this directory does not, however, have any read permissions set, and the file `user.new` is empty. These files are therefore retained in their original state.

```
$ ls -l GMC3
total 272
-rw-r--r--  1  naomi    gmc3    117865 Aug 19 09:27 data80-90
---x--x--x  1  naomi    gmc3      5366 Mar 15 12:50 print
-rw-r--r--  1  naomi    gmc3      1447 Jan 10 17:02 user.old
-rw-r--r--  1  naomi    gmc3         0 Mar 14 13:16 user.new
$ pack -f GMC3/*
  pack: GMC3/data80-90: 39.5% Compression
  pack: GMC3/print: cannot open
  pack: GMC3/user.old: 37.9% Compression
  pack: GMC3/user.new: cannot pack a zero length file
$ echo $?
2
$ ls -l GMC3
total 154
-rw-r--r--  1  naomi    gmc3     71311 Aug 19 09:27 data80-90.z
---x--x--x  1  naomi    gmc3      5366 Mar 15 12:50 print
-rw-r--r--  1  naomi    gmc3       899 Jan 10 17:02 user.old.z
-rw-r--r--  1  naomi    gmc3         0 Mar 14 13:16 user.new
```

SEE ALSO

compress(1), pcat(1), uncompress(1), unpack(1), zcat(1).

User's Reference Manual

page(1)

NAME

`page` - control the output on the screen

SYNOPSIS

`page` [−[+]aBcCdeEfFHiIlLmMqQrsuUw] [−b*number*] [−j*number*] [−[n]*number*] [−k*number*] [−I*number*] [−P[mM=]*string*] [−o *file* | −O *file*] [−K*file*] [−t*tag*] [−T*tagsfile*] [−p*command*] *file* ...

`page` −h

DESCRIPTION

`page` is identical to `more`. See `more`(1) .

User's Reference Manual

pagesize(1)
BSD Compatibility Package

NAME

`pagesize` - display the size of a page of memory

SYNOPSIS

`/usr/ucb/pagesize`

DESCRIPTION

`pagesize` prints the size of a page of memory in bytes, as returned by `getpagesize`. This program is useful in constructing portable shell scripts.

SEE ALSO

`getpagesize`(3).

User's Reference Manual

passwd(1)

NAME

passwd - define or change login password and password attributes

SYNOPSIS

passwd [*option ...*] [*login_name*]

DESCRIPTION

The passwd command can be used to define a password for a login name or change an existing password in the /etc/shadow file. It can also be used to list or change specific password attributes.

Any user may list the password attributes for his or her own login name. Ordinary users may change only the password and password attributes which correspond to their own login names. The system administrator, on the other hand, can change the passwords and password attributes for all local login names.

passwd prompts ordinary users (without system administrator privileges) for their old password, if one was defined.

It then prompts for the new password to be entered. To avoid errors, the new password has to be entered twice. The passwords are not displayed on the screen as they are typed.

When the old password is entered, passwd checks to see if it has "aged" sufficiently. If "aging" is insufficient, passwd terminates [see shadow(4)]. If no aging requirements have been defined for the user's password (options –n and –x), the default values MAXWEEK and MINWEEK from the /etc/default/passwd file are used [see default_passwd(4)]. If aging requirements have been defined, the corresponding information in the /etc/shadow file remains unmodified.

Assuming aging is sufficient, passwd checks that the new password has the correct format.

A password

- must be at least PASSLENGTH characters long, as defined in the /etc/default/passwd file. PASSLENGTH has a minimum value of six characters. passwd always evaluates only the first eight characters of a password, even if PASSLENGTH is greater than eight.
- must contain at least two alphabetic characters and at least one numeric or special character. The alphabetic characters may be entered in uppercase or lowercase.
- must differ from the user's login name and any reverse or circular shift of that login name. For comparison purposes, no differentiation is made between uppercase and lowercase.
- must differ from the old password (if one was defined) by at least three characters. For comparison purposes, no differentiation is made between uppercase and lowercase.
- must generally not contain the <CR> character.

If the ADDCHECK variable is set to yes in the /etc/default/passwd file [see default_passwd(4)], the following additional checks are carried out:

- A search is carried out for over-simplistic passwords such as a series of repeated letters.
- The password is compared with some host-specific information such as the host name.
- The password is checked to determine if it is formed from an arbitrary combination of letters in the user entry in /etc/passwd (inverting the login name, writing the login twice).
- The password is compared with information on the user at local and remote systems which the finger(1) command supplies.
- Dictionaries created specially [see makedict(1M)] are searched through to locate the password.

The checks carried out are not case-sensitive (except with character checks).

By default, the following characters are not permitted in passwords [see ascii(5)]:

CTRL-C (ETX), CTRL-D (EOT), CTRL-H (BS), CTRL-J (LF), CTRL-M (CR), CTRL-O (SI), CTRL-Q (DC1),

CTRL-R (DC2), CTRL-S (DC3), CTRL-Y (EM), CTRL-Z (SUB), CTRL-\ (FS), CTRL-[(ESC), CTRL-] (GS), DEL

The following types of password are accepted:

- Passwords containing uppercase and lowercase.
- Passwords containing punctuation marks.
- Passwords containing numbers.
- Passwords containing permitted CTRL characters.

The passwd command evaluates only the first 8 characters of a password.

The system administrator is not prompted for the old password, and no check is performed to see whether the new password is in the correct format (except for <CR>) or complies with password aging requirements.

OPTIONS

No option specified:

The password of the login name under which you logged in is changed.

−s Shows the password attributes for *login_name* in the following format:

login_name status MM/DD/YY minimum maximum warning

or, if no password aging requirements have been set:

login_name status

login_name

Login name under which the user has logged in.

status

The current password status of *login_name*: PS stands for passworded or locked, LK stands for locked, and NP stands for no password.

MM/DD/YY

The date on which the password for *login_name* was last changed. This date is indicated in Greenwich Mean Time and may therefore differ by as much as a day in other time zones.

minimum

The minimum number of days required between password changes for *login_name*. The default value is MINWEEKS, as defined in the /etc/default/passwd file.

maximum

The maximum number of days the password is to be valid for *login_name*. The default value is MAXWEEKS, as defined in the /etc/default/passwd file.

warning

The number of days before the password expires that the user will be warned.

The following options can only be used by the system administrator:

−l Locks the password entry for *login_name*.

−d Deletes the password for *login_name*. The *login_name* will not be prompted for a password.

Warning:

If the system administrator has deleted a password for a user using the −d option, and password aging is activated for that user, the user can no longer enter a new password even if the PASSREQ entry in /etc/login/default file is set to YES. This means that the user no longer has a password. For this reason you should always use the −f option with the −d option. This will force the user to change the password at the next login.

−n *min*

At least *min* days must pass between two password modifications. If *min* is greater than *max*, the user cannot change the password. You should always use this option with the −x option, unless *max* is set to −1 (aging turned off), in which case, *min* need not be set.

−x *max*
The password for *login_name* is valid for *max* days at the most. If *max* is set to −1, the aging for *login_name* is immediately turned off. If *max* is set to 0, aging is turned off and the user is forced to change the password at the next login (attention: this can only be carried out once within a 24-hour period).

−w *warn*
The user will be warned *warn* days before the password expires.

−a Lists the password attributes for all entries. You should use this option only in conjunction with the −s option. You need not specify the *login_name* in this case.

−f Forces the user to change the password at the next login by invalidating the password for *login_name*.

login_name
Is the login name to which the new password is to be allocated. This parameter is only useful for the system administrator. If you are not the system administrator, you can only specify your own login name.

login_name not specified:

`passwd` changes the password for the current login name.

The following option can be specified both by the system administrator and by a normal user:

−g (g – global) If this option is set, the new user password will be forwarded to all machines in domains managed by `SIclos` [see `clos(7)`] or WSA [see `wsa(1M)`]. Two cases can be distinguished here:

1. The option is invoked by the user `root`:

   ```
   passwd −g kurt
   ```

   The password is changed for the user `kurt` on all machines.

2. The option is invoked by a normal user `kurt`:

   ```
   passwd −g
   ```

   As usual, the old password of the user `kurt` is queried. The input is simply compared locally and the new password is then distributed to all machines in the domains even if the user `kurt` had a different password on these machines previously.

When the command terminates, the user receives a list of machines on which the password was changed successfully.

Warning:

If the `DOMAINPASS` variable [see `default_passwd(4)`] is not set and/or the `SIclos` package is not installed, `passwd` aborts with an error message.

EXIT STATUS

`0` Successful completion.

`1` Permission denied.

`2` Invalid combination of options.

`3` Unexpected failure. Password file unchanged.

`4` Unexpected failure. Password file missing.

`5` Password file in use. Try again later.

`6` Invalid argument to option.

ERROR MESSAGES

`passwd: xxx does not exist.`

There is no entry for this login name in the `/etc/passwd` file.

`passwd: Permission denied.`

You are not permitted to change the password for the specified login name. Only the system administrator is

permitted to change passwords for login names other than his/her own.

`Sorry.`

You did not enter the old password correctly when prompted for it.

`Password unchanged.`

The password could not be changed.

Password must contain at least two alphabetic characters and at least one numeric or special character.

The password must contain at least two non-numeric characters.

`Password is too short - must be at least 6 characters.`

You are required to enter a longer password.

`Too many failures - try later.`

You have made three unsuccessful attempts to change your password. The password entered by you does not meet the requirements. Enter a new password.

`Password must differ by at least 3 positions.`

Your new password must differ from the old one by at least three positions.

`password file busy - try again.`

The system administrator is currently working on the password file. To avoid inconsistencies, you must wait until this file is free again.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_TYPE` governs character classes and character conversion (shifting).

The `LC_TIME` environment variable governs the format of date and time strings.

If `LC_MESSAGES`, `LC_CTYPE` or `LC_TIME` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

It is advisable to restrict the characters you use for login names and passwords to those in the Portable Filename Character Set; 8-bit data may not always be portable to non-internationalized systems.

EXAMPLES

User `alibaba` working on system `thieves` wishes to change his password `sesame!street`. The new password is to be `open!sesame`:

```
$ passwd
passwd:  Changing password for alibaba
Old password:             Blanked input of sesame!street
New password:             Blanked input of open!sesame
Re-enter new password:    Repeated blanked input of open!sesame
```

FILES

`/etc/passwd`
: File in which user accounts/login names are entered.

`/etc/default/passwd`
: File containing the default settings for `passwd(1)`.

`/etc/default/checkpasswd`
: File containing additional settings for `passwd(1)`.

`/etc/shadow`
: "Hidden" password file containing the encrypted passwords. No general read access.

`/etc/.history/`*`login_name`*

The /etc/.history/*login_name* file is created the first time the user *login_name* is assigned a password with passwd(1). This file stores the encrypted password as many times as it is defined in the /etc/default/passwd file [see default_passwd(4)] with the HISTORY=*n* parameter.

SEE ALSO

id(1), login(1), su(1), makedict(1M), passmgmt(1M), pwconv(1M), useradd(1M), userdel(1M), usermod(1M), crypt(3C), checkpasswd(4), default_passwd(4), passwd(4), shadow(4).

User's Reference Manual

paste(1)

NAME

`paste -` merge same lines of several files or subsequent lines of onefile

SYNOPSIS

Format 1

`paste` [–`d` *list*] [––] *file ...*

Format 2

`paste –s` [`-d` *list*] [––] *file ...*

DESCRIPTION

`paste` can be used to horizontally merge the n-th corresponding lines from two or more input files (parallel merging, **Format 1**) or to successively merge all lines within a single file (serial merging, **Format 2**). The result is written to standard output.

OPTIONS

Format 1: Joining the n-th lines of several files (parallel merging)

`paste` [–`d` *list*] [––] *file ...*

`paste` concatenates the n-th line of each input file, treating each file as a column or columns of a table. The corresponding lines are pasted together horizontally and displayed on standard output (see *Example 1*).

No option specified:
The tab character acts as a delimiter between columns.

–`d` *list*
(d – delimiter) Uses a character from *list* as a delimiter between output columns.

The characters in *list* are used consecutively and circularly, i.e. `paste` returns to the top of the list after using the last character in it. In parallel merging, lines from the last input file are always terminated with a newline character, not with a character from the *list*.

Any string of arbitrary characters can be specified for *list*. The following escape sequences may also be used: `\n` (newline), `\t` (tab), `\\` (backslash), and `\0` (empty string, not a null character). If *list* contains escape sequences, blanks, or shell metacharacters, it must be enclosed in double quotes "...".

–– If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file Name of the input file. This format of `paste` is only meaningful if you specify several files.

If you use a dash (–) as the name for *file*, `paste` reads from standard input.

Format 2: Joining successive lines (serial merging)

`paste –s` [–`d` *list*] [––] *file ...*

–`s` (s – subsequent) For each input file, `paste` joins the lines together to form a single line and writes this line to standard output. By default, the lines from the input file are separated by tabs (see option `d`). Each output line is terminated by a newline character.

–`d` *list*
(d – delimiter) One of the characters from *list* is used in the output line instead of a tab to mark the joins between the input lines.

The characters in *list* are used consecutively and circularly, i.e. `paste` returns to the top of the list after using the last character in it.

Any string of arbitrary characters can be specified for *list*. The following escape sequences may also be used: `\n` (newline), `\t` (tab), `\\` (backslash), and `\0` (empty string, not a null character). If *list* contains

escape sequences, blanks, or shell metacharacters, it must be enclosed in double quotes "...".

–– If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file Name of the input file. You may name more than one file.

If you use a dash (–) as the name for *file*, `paste` reads from standard input.

ERROR MESSAGES

`paste: line too long`

Output lines must not exceed 511 characters.

`paste: too many files - limit 12`

A maximum of 12 input files may be specified in **Format 1**.

`paste: no delimiters`

You have not specified *list* with the –`d` option.

`paste: cannot open file`

file does not exist or you do not have read permission for *file*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Examples of Format 1

Example 1

The corresponding lines from the files `numbers` and `letters` are to be pasted together:

The `numbers` file contains numbers from 1 to 100:

```
  1
  2
  3
  .
  .
100
```

The `letters` file contains lowercase letters from `a` to `z`:

```
a
b
c
.
.
z
$ paste numbers letters
  1     a
  2     b
  3     c
  .     .
  .     .
 25     y
 26     z
 27
  .
  .
```

```
100
```

Example 2

The current directory contains the following files:

```
$ ls
corr
jokes
names
plan
probe
prog.c
tst
words
```

The following command numbers these files (see the `numbers` file in *Example 1*):

```
$ ls | paste numbers -
  1     corr
  2     jokes
  3     names
  4     plan
  5     probe
  6     prog.c
  7     tst
  8     words
  9
 10
  .
  .
100
```

See also *Example 3*.

Example 3

The current directory contains the same files as in *Example 2*. The following command lists the contents of the current directory in three columns. However, the columns will only be justified properly if the individual file names do not extend beyond the next tab stop.

```
$ ls | paste - - -
corr    jokes   names
plan    probe   prog.c
tst     words
```

How is this output produced? Compare the above command with the command

```
$ paste file1 file2 file3
```

In this case, `paste` first reads the initial lines from all three files and pastes them into one line. When this is done, the second lines are read, etc.

In the command `ls | paste - - -`, the first file name that `paste` reads from standard input (`corr`) corresponds to the first line from `file1`; the second file name `jokes` corresponds to the first line from `file2` etc.

Example 4

The current directory contains the same files as in *Example 2*. As in *Example 3*, you now wish to display the file names in three columns, but the second and third columns are to be separated by a colon instead of a tab.

```
$ ls | paste –d "\t:" - - -
corr    jokes:names
plan    probe:prog.c
tst     words:
```

Example of Format 2

Example 5

The `customers` file contains the following:

```
hansen
smith
cologne
koch
schulz
london
tornio
meyer
perth
$ paste -s customers
hansen  smith   cologne koch    schulz  london  tornio  meyer   perth
```

The following command joins only three lines of the `customers` file at a time. This is because a newline character is specified as a delimiter after every third input line:

```
$ paste –s –d "\t\t\n" customers
hansen  smith   cologne
koch    schulz  london
tornio  meyer   perth
```

SEE ALSO

`cut`(1), `grep`(1), `pr`(1).

User's Reference Manual

patch(1)

NAME

`patch` - use diff list

SYNOPSIS

`patch` [*option ...*] [*file*] [+ [*option ...*] [*file*]]

DESCRIPTION

`patch` will take a patch file containing any of the three forms of difference listing (normal, with context or in the `ed` format), produced by the `diff(1)` program and apply those differences to an original file, producing a patched version. By default, the patched version is put in the place of the original. If you specify the `–b` option, the original file is stored under the same name with the extension `.orig`. You may also specify where you want the output to go with an `–o` option.

Upon startup, `patch` will attempt to determine the type of the diff listing, unless overruled by a `–c`, `–e` or `–n` option. Context diffs and normal diffs are applied by the `patch` program itself, while `-e` diffs are simply fed to the `ed(1)` editor via a pipe.

`patch` will try to skip any leading extraneous material, apply the diff, and then skip any trailing extraneous material. If the entire diff is indented by a consistent amount, this will be taken into account.

With context diffs, and to a lesser extent with normal diffs, `patch` can detect when the line numbers mentioned in the patch instructions are incorrect, and will attempt to find the correct place to apply each hunk of the patch. As a first guess, `patch` takes the line number mentioned for the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct position, `patch` will scan a maximum of 1,000 bytes forwards and backwards for a set of lines matching the context given in the hunk. First `patch` looks for a place where all lines of the context match. If no such place is found, and it is a context diff, and the maximum fuzz factor is set to 1 or more, then another scan takes place ignoring the first and last line of context. If that fails, and the maximum fuzz factor is set to 2 or more, the first two and last two lines of context are ignored, and another scan is made. (The default maximum fuzz factor is 2.)

If `patch` still cannot find a place to install that hunk of the patch, it will write the hunk to a reject file, which normally is the name of the output file plus `.rej`. Note that the rejected hunk will come out in context diff form whether the input patch was a context diff or a normal diff. If the input was a normal diff, many of the contexts will simply be null.) The line numbers on the hunks in the reject file may be different than those in the patch file: they reflect the approximate location `patch` thinks the failed hunks belong in the new file rather than the old one.

As each hunk is completed, you will be told whether the hunk succeeded or failed, and in which line (in the new file) `patch` thought the hunk should go. If this is different from the line number specified in the diff, you will be told the offset. A single large offset may be an indication that a hunk was installed in the wrong place. You will also be told if a fuzz factor was used to make the match, in which case you should also be careful.

If `patch` is called with an `ed` list, this list is forwarded to the `ed(1)` editor. Incorrect `ed` instructions cannot be diagnosed with the result that no `.rej` error file is created and the status 1 is not returned (see **EXIT STATUS**).

If no original file is specified, `patch` will try to figure out from the leading extraneous material what the name of the file to edit is. In the header of a context diff, the filename is found from lines beginning with "***" or "---". If `patch` cannot determine the filename in this way, it searches for an "Index:" line and attempts to use the filename contained in this line. Filenames found are then truncated in accordance with the `–p` option. `patch` searches the file(s) in the current directory (or the directory specified with the `–d` option). If the original file cannot be found, and a matching SCCS or RCS file is available, `patch` tries to retrieve and edit this file. If no filename can be determined from the leading extraneous material, you will be asked for the name of the file to patch.

If the leading extraneous material contains a "Prereq:" line, `patch` will take the first word from the

prerequisites line (normally a version number) and check the input file to see if that word can be found. If not, `patch` will ask for confirmation before proceeding.

If the patch file contains more than one patch, `patch` will try to apply each of them as if they came from separate patch files. For every patch, `patch` tries to determine the name of the file to be corrected and scans the leading and trailing extraneous material for relevant specifications (like for example, filenames and revision level). You can specify options (and another original filename) for the second or subsequent patches by separating the corresponding argument lists with a "+".

OPTIONS

The options `–c`, `–e` and `–n` cannot be combined.

`–b` Saves all original files with a `.orig` suffix before the patch is applied. The backup file is overwritten if it already exists; if patch is executed several times, a backup file is only created on the first execution. The additional option `–o` does not save the original file, but saves the output file if it already exists.

`–B` *backup*
Saves the original file in *backup*`.orig` before the patch instructions are executed.

`–c` Interprets the patch file as a context diff (output from `diff`, if `–c` or `–C` is specified).

`–d` *directory*
`patch` changes to the *directory* directory before further actions take place.

`–D` *define*
Marks changes with the C preprocessor construct:

```
#ifdef define
...
#endif
```

The argument *define* will be used as the differentiating symbol.

Note that, unlike with the C compiler, there must be a space between the `–D` option and the argument.

`–e` Interprets the patch file as an `ed` script.

`–f` `patch` assumes that the user knows exactly what he or she is doing, and asks no questions. It does not suppress commentary, however. Use `–s` for that.

`–F` *lines*
Sets the maximum fuzz factor. This option only applies to context diffs, and causes `patch` to ignore up to that many lines in looking for places to install a hunk. Note that a larger fuzz factor increases the odds of a faulty patch. The default fuzz factor is 2, and it may not be set to more than the number of lines os context in the context diff, ordinarily 3.

`–i` *patchfile*
`patch` reads the patch from the *patchfile* file. If you enter a dash for *patchfile* `patch` reads from the standard input.

A patch file contains one or more patches and maybe some additional information. If a patch file contains several patches, each patch should contain information about filenames (like with `diff –c`) so that `patch` can find affected files automatically.

`patch` evaluates the following information:

`Index:` *pathname*
pathname names the file to be corrected

`***` *pathname*
pathname specifies the "old" file on which the patch is based

`---` *pathname*
pathname specifies the file to be corrected (has priority over `Index:`)

Each patch contains patch instructions, which correspond to one of the three kinds of diff.

–i not specified: patch reads from the standard input.

–l Any sequence of tabs and blanks in the patch file will match any sequence of tabs and blanks in the input file. However, normal characters must still match exactly.

–n Interprets the patch file as a normal diff.

–N Ignores patches that have already been applied. Such patch instructions are rejected by default.

–o *output-file*
The corrected version is written to *output-file*. Several corrected files are appended one after the other. If different patches affect the same original file, the following scripts are applied to a temporary file. Several versions of the original file are then written to *output-file* accordingly.

–p *number*
Controls the handling of pathnames found in the patch file. *number* specifies how many slashes are to be stripped from the front of the pathname. (Any intervening directory names are also removed.) For relative pathnames, the search takes place in the current directory or in the directory specified by the –d option.

–p not specified: Only the basic name, without path, is used.

For example, supposing the filename in the patch file was

```
/u/howard/src/blurfl/blurfl.c
```

Setting the `–p 0` gives the entire pathname unmodified, while `–p 1` gives

```
u/howard/src/blurfl/blurfl.c
```

without the leading slash, `–p 4` gives

```
blurfl/blurfl.c
```

and not specifying –p at all just gives you `blurfl.c`.

–r *rejectfile*
Specifies the file to which the rejected patchs should be sent.

–r not specified:

The rejected files are sent to one file, which has the same name as the output file, but with the suffix `.rej`.

–R patch swaps the patch instructions before they are applied to the original file. This is necessary if the old and new files were exchanged when creating the patch. patch attempts to swap each hunk before it is applied. Rejects are written to the error file in swapped format. The –R option cannot be used for ed scripts, as the information is insufficient for reconstructing the swapped operation.

If the first hunk of a patch fails, patch swaps the patch instructions to see if they can be applied in this way. If this is possible, you are asked whether the –R option should be set. If this is not possible, patch enters the hunk in the error file and continues as normal. (*Note*: reversed patch instructions cannot be detected with this method in the case of a normal diff and where the first command is an append, i.e. it should in fact have been a delete.)

–s patch does not output any messages if no errors occur.

–S If the patch file contains several patches, a script can be skipped using –S. The next script is then applied to the original file.

```
patch -S + -S + file
```

ignores, e.g. the first two of approximately three patches.

–v Outputs information on the version of the active patch program.

–x *flag*
Sets internal debugging flags. This option can be used for testing the patch file.

file Pathname of the file to be patched.

file not specified:

`patch` attempts to determine the filename from the leading extraneous material.

NOTES FOR PATCH SENDERS

If you create patch files on a regular basis, it is recommended that the revision level be included as the first patch instruction. If you add the line "Prereq: " to the patch file, you can prevent users applying incorrect patches without being warned.

Check whether the filenames have been specified correctly in the context diff header or in the "Index:" line. If you wish to make corrections in a subdirectory, inform the patch user that the `-p` option must be set.

Avoid reversed patches where possible.

Place patches that need to be together in the event of an error in separate files where possible.

If patch files are created with rejected hunks, `patch` terminates with a non-zero exit status. If you want to apply a set of patches in a loop, you should check this exit status so that a subsequent patch will not be performed on a partly corrected file.

If code was duplicated (e.g. using "`#ifdef OLDCODE ... #else ... #endif`"), `patch` cannot correct both versions. If the command can be executed at all, it may correct the wrong version and inform you that the corrections have been made successfully.

If you use an already applied patch again, `patch` assumes that the patch in question is a reversed patch and offers to un-apply the patch.

ENVIRONMENT VARIABLES

The following environment variables have an effect on the execution of `patch`:

`LANG` Specifies a default value for the locale variables that are unset or null. If `LANG` is unset or null, the corresponding locale default value is used. If the locale variable contains an invalid setting, `patch` behaves as if no variables had been set.

`LC_ALL`
If this value is set, i.e. is not empty, this value overwrites the values of all other locale variables.

`LC_CTYPE`
Determines the locale for the interpretation of byte sequences as characters (e.g. single-byte as opposed to multibyte characters in arguments).

`LC_MESSAGES`
Determines the format and content of error messages.

`NLSPATH`
Determines the position of the message catalog for the processing of `LC_MESSAGES`.

EXIT STATUS

`0` Successful completion.

`1` One or more rejected patch instructions were written to the error file.

`>1` An error occurred.

FILES

`/tmp/patch*`

SEE ALSO

`diff(1)`, `ed(1)`.

User's Reference Manual

pathchk(1)

NAME

`pathchk` - check pathnames

SYNOPSIS

`pathchk` [`-p`] *pathname* ...

DESCRIPTION

The `pathchk` command checks that one or more pathnames are valid (i.e. they can be used to access or create a file without causing syntax errors) and portable (i.e. the name need not be adjusted). More extensive portability checks can be carried out with the `-p` option.

By default, the `pathchk` command checks the components of all *pathname* arguments based on the underlying file system. An error message is output for the *pathname* argument if:

- it is longer than the maximum permitted pathname length (`PATH_MAX` bytes).
- it contains a component that is longer than the maximum permitted filename length (`NAME_MAX` bytes) in the relevant directory.
- it contains a component in a directory that is not searchable.
- it contains a component with characters that are invalid in the directory.

`pathchk` does not consider it an error if one or more components of a `pathname` argument do not exist, as long as the file with the specified pathname can be created and does not violate any of the checks described above.

OPTIONS

`-p` `pathchk` does not check the underlying file system, rather the generic portability conditions. An error message for the *pathname* argument is output if:

- it is longer than the maximum permitted length for portable pathnames (`_POSIX_PATH_MAX` bytes).
- it contains a component longer than the maximum length for portable filenames (`_POSIX_NAME_MAX` bytes).
- it contains a component with characters not contained in the portable character set for filenames.

pathname
The pathnames to be checked.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of `pathchk`:

`LANG` Specifies the default value for the variables for the locale that is unset or null. If `LANG` is unset or null, the corresponding default value of the locale is used. If the internationalization variable contains an invalid setting, the command behaves as if none of the variables was set.

`LC_ALL`
If this value is set, i.e. is not empty, this value overwrites the values of all the other variables for the locale.

`LC_CTYPE`
Determines the interpretation of byte sequences as characters (e.g. as single-byte as opposed to multi-byte characters in arguments).

`LC_MESSAGES`
Determines the format and content of messages.

EXIT STATUS

`0` All pathnames are error-free.

`>0` An error occurred.

DIAGNOSTICS

`pathchk: Subpath` *path* `is not searchable`
`ENOSYS` error reported by the `pathconf()` system call [see `fpathconf(2)`].

APPLICATION USAGE

You can use the `test` command to check whether a pathname specifies an existing file. However, this does not say whether a component of the pathname was truncated (in a directory where the `_POSIX_NO_TRUNC` function is not activated). The `pathchk` command does not check for the existence of a file. It only checks if a certain pathname exists or if it can be created without truncating the name.

The `noclobber` option in the shell [compare with the description of `set(1)`] creates a unique file.

EXAMPLES

This is how you check if all pathnames in an imported data interchange archive are legitimate and unambiguous on the current system:

```
pax -f archive | sed -e '/ == .*/s///' | xargs pathchk
if [ $? -eq 0 ]
then
    pax -r -f archive
else
    echo Investigate problems before importing files.
    exit 1
fi
```

This is how you check whether all files in the current directory hierarchy can be transferred to another system, that the general portability conditions are met and that the `pax(1)` command is available:

```
find . -print | xargs pathchk -p
if [ $? -eq 0 ]
then
    pax -w -f archive .
else
    echo Portable archive cannot be created.
    exit 1
fi
```

You can check whether a specified pathname names a readable file and whether an application can create a file by extending the filename without truncating the pathname and without overwriting any existing files.

```
case $- in
    *C*)    reset="";;
    *)      reset="set +C"
            set -C;;
esac
test -r "$path" && pathchk "$path.out" &&
    rm "$path.out" > "$path.out"
if [ $? -ne 0 ]; then
    printf "%s: %s not found or %s.out fails \
            creation checks.\n" $0 "$path" "$path"
    $reset  # reset the noclobber option in case a trap
            # on EXIT depends on it
    exit 1
fi
$reset
PROCESSING < "$path" > "$path.out"
```

This example is based on the following:

1. `PROCESSING` displays the code used by the application in order to use `$path`, once it has been verified that `$path.out` meets the required conditions.
2. The state of the `noclobber` option is unknown when this code is called, and should be reset on exit to the state it was in when the code was called. (The `reset` variable restores the initial state in this example.)
3. Note the usage of the following construction:

   ```
   rm "$path.out" > "$path.out"
   ```

   a) The `pathchk` command has already checked that `$path.out` is not truncated.

   b) If the `noclobber` option is set, the shell checks that `$path.out` does not already exist before `rm` is called.

   c) If the shell has successfully created `$path.out`, `rm` removes it again so that the application can create the file in the `PROCESSING` step.

   d) If the `PROCESSING` step wants the file to exist already when it is called,

   ```
   rm "$path.out" > "$path.out"
   ```

   should be replaced by

   ```
   > "$path.out"
   ```

   This confirms that `$path.out` does not yet exist and will be created for `PROCESSING`.

SEE ALSO

`test`(1), `fpathconf`(2).

User's Reference Manual

pax(1)

NAME

pax - portable archive processing

SYNOPSIS

pax [−cdnv] [−f *archive*] [−s*instruction*] ... [*pattern* ...]

pax −r[−cdiknuv] [−f *archive*] [−p*string*] ... [−s *instruction*] ... [*pattern* ...]

pax −w[−dituvX] [−b *blocksize*] [−a] [−f *archive*] [−s *instruction*] ... [−x *format*] [*file* ...]

pax −r −w[−diklntuvX] [−p *string*] ... [−s *instruction*] ... [*file* ...] *directory*

DESCRIPTION

The pax command writes files to archive files, reads them or creates lists of these files and copies directory hierarchies. A number of archive formats are supported; see the −x *format* option.

The action executed depends on which of the −r and −w options are specified. The four possible combinations of −r and −w give the four operating modes: list mode, read mode, write mode, and copy mode, corresponding respectively to the four forms shown in the **SYNOPSIS**.

list In list mode (i.e. neither −r nor −w is specified), pax reads an archive file from the standard input and writes the name of the stored files, whose pathnames match the specified *pattern*, to standard output. If a specified file is a directory, the file hierarchy rooted at that file will be written out also.

read In read mode (i.e −r is specified, but not −w), pax reads an archive file from the standard input and extracts the files whose pathnames match the specified *pattern*. If the extracted file is a directory, the file hierarchy rooted at that file is also extracted. The extracted files are created relative to the current directory.

The ownership, access and modification dates and times, as well as file mode of the restored files are described in the −p option.

write

In write mode (i.e. −w is specified, but not −r), pax writes the contents of the *file* operands to standard output in archive format. If no *file* operand is specified, a list of files to copy, one per line, is read from the standard input. When specifying a directory, all files in the file hierarchy rooted at that file are copied.

copy In copy mode (i.e. both −r and −w are specified), pax copies the *file* operands to the destination directory *directory*.

If no *file* operands are specified, a list of all files to be copied, one per line, is read from the standard input. When specifying a directory, all files in the file hierarchy rooted at that file are copied.

The copying process runs as if the copied files were written to an archive file and subsequently extracted, except that there may be hard links between the original and the copied files.

If temporary directories are needed to extract a file in read or copy mode, these are created by pax.

At least one file must match the specified *pattern* or *file* operands, otherwise pax writes a diagnostic message to the standard message output for each operand that is unmatched, and terminates with a non-zero exit status.

The archive formats supported are automatically recognized when reading the file. The standard output format for archives when writing (no −x option specified) is the extended tar format.

An archive can span multiple files. The pax command determines what file to read or write as the next file.

If the selected archive format supports the specification of links between files, pax returns an error if no link can be created between these files during extraction.

OPTIONS

`-r` Reads an archive file from the standard input.

`-w` Writes files in the specified archive format to the standard input.

`-a` Appends files to the end of the archive.

`-b` *blocksize*
Puts the data in blocks in the archive file, and specifies the block size as a positive decimal integer. Devices and archive formats may impose restrictions on blocking. Blocking is automatically determined on input. Default blocking during the creation of archives depends on the archive format (see the `-x` option below). To maintain a portable archive, the block size may not be bigger than 32 KB.

`-c` Selects all files apart from those specified by the *pattern* or *file* operands.

`-d` The file hierarchy rooted at this file in a directory is not copied, read, or archived.

`-f` *archive*
Specifies the pathname of the input archive (in list or read mode) or the output archive (write mode). This archive is used instead of the standard input or output.

`-i` Renames files interactively. For every file archived whose name matches the *pattern* operand, or for every file that matches a *file* operand, a prompt is written to `/dev/tty`. This prompt contains the name of the file. A line is read from `/dev/tty`. If this line is blank, the file is skipped. If the line consists of a single period, the file is processed without the name being modified. Otherwise the name is replaced by the contents of the line. If an end-of-line character is found when reading a response, or if `/dev/tty` cannot be opened for reading or writing, the `pax` command terminates immediately with a non-zero exit status.

`-k` Prevents the overwriting of existing files.

`-l` Creates links between files. Hard links are created whenever possible between source and target file hierarchies.

`-n` Selects the first file archived that matches the *pattern* operand. Only one matching file is selected for every *pattern* operand (file hierarchies rooted at the directories are not affected by this restriction).

`-p` *string*
Determines file characteristics (privileges). The *string* argument contains the file characteristics that must be retained or ignored during extraction. The *string* consists of the `a`, `e`, `m`, `o` and `p` specification characters. Several characters can be concatenated in the same string, and the `-p` option can be specified more than once. The specification characters mean the following:

- `a` Date and time of the last file access are set to the current date.
- `e` User ID, group ID, access permissions (see `chmod`), date and time of the last access and modification are preserved.
- `m` Date and time of the last file modification are not preserved.
- `o` User ID and group ID are preserved.
- `p` Access rights are preserved.

In the list above, "preserved" means that an attribute saved in the archive will be assigned to the extracted file, depending on the privileges of the invoking process. Otherwise, the attribute is defined as part of the normal file creation action.

`pax` does not set the s bits for the file mode if neither `e` nor `o` are specified, or the user and group IDs preserved.

If for any reason, one of these attributes cannot be preserved, `pax` writes a diagnostic message to standard output. If an attribute cannot be preserved, the extracted file is not deleted, although it does effect the exit status.

If specification characters are duplicated in the *string* argument or are in conflict with the other arguments, the last argument entered has priority. If for example `-p eme` is specified, the date and time of the last file modification are preserved.

−s *instruction*

Modifies files specified by the *pattern* or *file* operands, according to the *instruction*. The syntax of ed is used for this. The concepts "address" and "line" mean nothing in the context of the pax command. The following format is used:

−s /old/new/[gp]

As in ed, *old* is a simple regular expression and *new* can contain & signs, *n* (where *n* is a number), references, and subexpressions. The *old* string can also contain a newline character.

All characters can be used as delimiters (e.g. /). A number of −s expressions are allowed. The expressions are evaluated in the specified order ending with the first successful substitution. The optional trailing g is as defined in ed. The optional trailing p writes successful substitutions to the standard error output. Filenames replaced by a null string are ignored when reading and writing archives.

−t The date and time of the last access to archived files are set to the value they had before being accessed by pax.

−u Ignores files that are an older version (with an older last modification date/time) of a file of the same name that already exists. In read mode, an archived file that has the same name as a file in the file system is extracted if the archived file is more current than the one in the file system. A file is only saved in write mode if there is no more recent version with the same name.

Note: pax −wu behaves in the same manner as −wau, i.e. files are appended to the end of the archive.

In copy mode, a file in the target hierarchy is replaced by a file in the source hierarchy by a link to the file in the source hierarchy, if it is the more current.

−v In list mode, a detailed table of contents is written to the standard output. Otherwise the pathnames of the stored file are written to standard output.

−x *format*

Specifies the archive output format. pax recognizes the following formats:

cpio The extended cpio interchange format. The default block size for this character-oriented archive file format is 5120. Implementations of pax support all values for the block size up to 32256 that are a multiple of 512.

Note: pax only displays transportable cpio archives. These are generated with cpio −o −H odc.

ustar

The extended tar interchange format. The default block size for this character-oriented archive file format is 10240. Implementations of pax support all values for the block size up to 32256 that are a multiple of 512.

Note: You must use this format for files larger than 2□GB. A warning is issued in this case.

pax terminates immediately with a non-zero exit status if an attempt is made to append data to an archive file in a different format than the existing archive format.

−X pax will not switch to a directory in a different file system when traversing a file hierarchy specified by a pathname.

The options that operate on the names of files (−c, −i, −n, −s, −u and −v) are evaluated one after the other as described below:

In write mode, the files are selected according to pathnames specified by the user, and the modifications performed using the −n and −u options. The −s and −i options then modify the names of the selected files. The −v option outputs the names resulting from these modifications.

If both the −u option and the −n option are specified, px does not consider a file unless it is more current than the file to which it is compared.

directory

Pathname of the target directory for copy mode.

file Pathname of the file to be copied or archived.

pattern
A pattern that matches one or more pathnames of archived files. By default (if no *pattern* is specified), all files in the archive are selected.

STDIN

In write mode, the standard input is only used if no *file* operands are specified. Standard output must then be a text file containing a list of the pathnames. The file must contain one pathname per line, and there may be no preceding or trailing blank characters.

In list and read mode, the standard input must be an archive file.

Otherwise the standard input is not used.

INPUT FILES

The input file specified by *archive* or the standard input if the archive is read from it, is a file formatted according to one of the archive formats listed in the `-x` option.

Prompts are written and responses read to or from `/dev/tty`.

ENVIRONMENT VARIABLES

`LANG` Specifies a default value for the locale variables that are unset or null. If `LANG` is unset or null, the corresponding default value of the locale is used. If the locale variable contains an invalid setting, `pax` behaves as if none of the variable was set.

`LC_ALL`
If this variable has a value, i.e. is not empty, this value overwrites the values of all other locale variables.

`LC_COLLATE`
Determines the locale for the behavior of ranges, equivalence classes and collating elements used in the pattern matching expressions for the *pattern* operand, in the simple regular expression for the `-s` option, and in the extended regular expression defined for the `yesexpr` locale keyword in the `LC_MESSAGES` category.

`LC_CTYPE`
Determines the locale for the interpretation of byte sequences (e.g. single-byte as opposed to multibyte characters in arguments and input files), for the behavior of character classes used in the extended regular expressions defined for the `yesexpr` locale keyword in the `LC_MESSAGES` category, and for pattern matching.

`LC_MESSAGES`
Determines the locale for the processing of yes/no responses, as well as the format and language of diagnostic messages, output by `pax`.

`LC_TIME`
Determines the format and the content of date and time specifications if the `-v` option is specified.

`NLSPATH`
Determines the position of the message catalog for the processing of `LC_MESSAGES`.

STDOUT

If `-f` is not specified in write mode, standard output is the archive formatted according to one of the archive formats listed in the `-x` option. □
In list mode, the table of contents of the selected files are written to standard output in the following format:

`"%s\n",` *pathname*

If `-v` is specified in list mode, the table of contents of the selected files is written to standard output in the following format:

Pathnames that represent hard links to previous files are written in this format:

`"%s == %s\n", ls_-l_listing, linkname`

All other pathnames are written in this format:

`"%s\n", ls_-l_listing`

ls_-l_listing is the format created by the `ls` command with the `-l` option.

STDERR

If `-v` is specified in read, write or copy mode, `pax` writes the pathnames it processes to the standard error output in the following format:

`"%s\n", pathname`

These pathnames are output at the start of file processing. The final newline character is output after the file is read or written.

If the `-s` option is specified, and there is a trailing `p` in the substitution string, substitutions are written to the standard error output in the following format:

`"%s >> %s\n", original_pathname, new_pathname`

Any messages about the archive format are also written to the standard error output.

OUTPUT FILES

In read mode, the files extracted or copied are of the archived file type.

In write mode, the output file named by the `-f` option is a file formatted according to archive formats listed in the `-x` option.

EXIT STATUS

`0` All files were successfully processed.

`>0` An error occurred.

CONSEQUENCE OF ERRORS

If `pax` cannot create a file or a link when reading an archive, cannot find a file when writing an archive, or cannot preserve the user ID, group ID, or access rights with the `-p` option set, a diagnostic message is written to the standard error output and a non-zero exit status is returned. Processing is continued nonetheless. If `pax` cannot create a link to a file, by default it does not create a second copy of the file.

If the extraction of a file from an archive is prematurely ended by a signal or an error, `pax` may only have extracted a part of the file, or (if the `-n` option is not specified) a file under the name specified by the user, that is not the file the user wanted.

Extracted directory access rights may also have additional bits from the file-creation mode mask as well as incorrect information about the date and time of the last file access and modification.

APPLICATION USAGE

The `-p` option was introduced to reconcile differences between the conventional `tar` and `cpio` implementations. In particular these two commands use `-m` in very different ways. The `-p` option also provides extended facilities for the consistent addressing of future file attributes, e.g. for extended security systems or high performance archives. Of the many combinations, only two modes are usually used:

`-p e` "Preserve everything" is used by the conventional superuser with all the appropriate privileges to preserve all file attributes as they are recorded in the archive. The `e` flag is the sum of `o` and `p`.

`-p p` "Preserve" the access rights. This is used by users with regular privileges who would like to preserve all file attributes apart from the owner. The file date and time specifications are preserved by default. However this can be deactivated with two other flags, and the date and time of extraction used.

Some of the functionalities described require appropriate privileges for whoever calls `pax`, particularly when creating block-oriented or character-oriented special files, when restoring the date and time of file access (`-t`

option), provided that the user is not the owner of the file, or when preserving the group, and the mode (`-p` option).

MESSAGES

```
Warning: just archiving file >2GB.
Unable to handle file >2GB with "-x cpio" option.
```

EXAMPLES

The following command writes all files as well as the contents of directories whose names begin with "g" to the archive `file.pax` (pack):

```
pax -wv -f file.pax g*
```

The following command reads the archive `file.pax` (unpack):

```
pax -rv -f file.pax
```

The following command copies the contents of the current directory to a tape:

```
pax -w -f /dev/rmt/m0 .
```

The following commands copy the `olddir` file hierarchy to `newdir`:

```
mkdir newdir
pax -rw olddir newdir
```

The following command reads the `a.pax` archive, whereby all file directories/files rooted in `/usr` in the archive relative to the current directory are extracted.

```
pax -r -s ',//*usr//*,,' -f a.pax
```

SEE ALSO

`cpio(1)`, `tar(1)`.

User's Reference Manual

pcat(1)

NAME

`pcat -` output packed files in unpacked form

SYNOPSIS

`pcat` *file* ...

DESCRIPTION

`pcat` takes files that have been compressed with `pack` and copies them to standard output in their original form. The packed file remains packed. Files compressed with `pack` can be restored to their original form with the `unpack` command. Unlike `cat(1)`, `pcat` is not a filter, since it does not read from standard input.

ARGUMENTS

file Name of the packed file to be output in its original form. You can name any number of files.

You may enter the name of the file with or without the `.z` extension; if you omit the extension, `pcat` automatically searches for the appropriate `.z` file.

ERROR MESSAGES

All the errors described below result in the failure of the `pcat` command.

`pcat: `*very_long_file_name*`: file name too long`

The name of the compressed file with `.z` must not exceed the maximum possible name length in the file system (`NAME_MAX`).

`pcat: `*file*`: cannot open`

You have no read permission for *file*, or *file* does not exist.

`pcat: `*file*`.z: not in packed format`

Although *file*`.z` has the `.z` suffix, it is not the result of a previous `pack` call.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The file `readme` is printed with `cat`, compressed with `pack`, and then output again with `pcat`. Since `readme` is too small for compression to be meaningful, `pack` is invoked with the `–f` option.

```
$ cat readme
I have still not been compressed!

$ pack –f readme
pack: readme: –47.1% Compression

$ pcat readme.z
I have still not been compressed!
```

Example 2

An uncompressed copy of the packed file `comp.z` is to be made under the name `uncomp`.

```
$ pcat comp >uncomp
```

SEE ALSO

cat(1), pack(1), unpack(1), zcat(1).

User's Reference Manual

pfmt(1)

NAME

`pfmt -` display error message in standard format

SYNOPSIS

`pfmt` [`-l` *label*] [`-s` *severity*] [`-g`*catalog*`:`*msgid*] *format* [*args*]

DESCRIPTION

`pfmt` uses *format* for `printf` style formatting of *args*. If the `-g` option is specified, `pfmt` retrieves a localized version of the *format* string from a locale-specific message database. The output is displayed on standard error (stderr).

`pfmt` encapsulates the output in the standard error message format. The `-l` *label* option specifies the label string to be displayed with the message (e.g. `"UX:cat"`).

label is a character string no more than 25 characters in length; it will be automatically suffixed with a colon (`:`). When unspecified, no label is displayed as part of the message.

The environment variable `LC_ALL_LABELS` is used to determine if labels will be added to the text of the messages. The value of `LC_ALL_LABELS` is consulted during the formation of the first message. Labels will be added (or not added) to all subsequent messages in the same way. Changing or setting `LC_ALL_LABELS` after the first message will have no effect message formation within the same execution.

If `LC_ALL_LABELS` is set to the value 1, the label string will be included with the message. If `LC_ALL_LABELS` does not exist, or is empty, or is set to 0, only the text of the message is displayed. Any other values of `LC_ALL_LABELS` will produce undefined results.

For example, consider the following message produced when `LC_ALL_LABELS` is set to 1:

```
UX:cat: ERROR: Cannot open door: No such file or directory
```

If `LC_ALL_LABELS` had not existed, or was empty, or had been set to 0, the above message would have been:

```
cat: Cannot open door: No such file or directory
```

The `-s` *severity* option specifies the severity string to be displayed with the message. Acceptable strings include the standard severities in either their print string (i.e. `HALT`, `ERROR`, `INFO`, `WARNING`, and `"TO FIX"`) or keyword (i.e. `halt`, `error`, `info`, `warn`, and `action`) forms, or any other user-defined string. A user-defined string will be assigned the integer severity value of 5. The severity will be suffixed with a colon (`:`). The `ERROR` severity will be used if no severity is specified.

The `-g` *catalog*`:`*msgnum* option specifies that a localized version of the *format* should be retrieved from a message database.

catalog is used to indicate the message database that contains the localized version of the *format* string. *catalog* must be limited to 14 characters. These characters must be selected from a set of all characters values, excluding `\0` (null) and the ASCII codes for `/` (slash) and `:` (colon).

msgnum is a positive number that indicates the index of the string in the message database.

If the catalog does not exist in the current locale (identified by the `LC_MESSAGES` or `LANG` environment variables), or if the message number is out of bounds, `pfmt` will attempt to retrieve the message from the C locale. If this second retrieval fails, `pfmt` uses the *format* string as passed on the command line.

`pfmt` will output `Message not found!!\n` as the *format* string if *catalog* is not a valid catalog name, or if *msgnum* is not a valid number.

STANDARD ERROR MESSAGE FORMAT

`pfmt` displays error messages in the following format:

label`:` *severity*`:` *text*

If no *label* is defined using the `-l` *label* option, the message is displayed in the format:

severity`:` *text*

If `pfmt` is called twice to display an error message and a helpful action or recovery message, the output can look like:

label`:` *severity*`:` *text*
label`:` `TO FIX:` *text*

EXIT STATUS

Upon success, `pfmt` exits with code 0. Upon failure, `pfmt` exits with the following codes:

`1` write error.

`3` syntax error.

EXAMPLE

```
pfmt -l UX:test -s error "Syntax error\n"
```

displays the message:

```
UX:test: ERROR: Syntax error
```

SEE ALSO

`fmtmsg`(1), `gettxt`(1), `lfmt`(1), `printf`(1), `environ`(5).

User's Reference Manual

pg(1)

NAME

pg - file perusal filter

SYNOPSIS

pg [*–number*] [–p *string*] [*option ...*] [*file_name ...*]

DESCRIPTION

pg writes files to the standard output. If the standard output is a terminal, you can view the files by scrolling backward or forward one page at a time.

pg displays a colon (:) as the default prompt at the bottom of each screen. You respond to this prompt by entering a command. The command you select determines which page is displayed next (see **COMMANDS DURING A PG SESSION**).

A character that follows a tab is displayed at the next tab stop; the tab spacing is assumed to be eight characters.

If the standard output is not a terminal, the file is not subdivided into pages, and no commands may be entered. pg then acts like the cat utility, except that if there is more than one file, pg prints a header for each file to identify it.

To determine terminal attributes, pg scans the terminfo database for the terminal type specified by the environment variable TERM. If TERM is not defined, the terminal type dumb is assumed.

OPTIONS

No argument specified:

pg reads from the standard input and writes this input a page at a time to the standard output.

–number

Specifies the number of screen lines to be treated as a page. The last line of one pageful is made the first line of the next.

Unless you use the –f option, pg splits lines longer than the screen width. Each line on the screen is thus treated as a separate line.

number

Number of lines to be used.

number=1: same as *number*=2; thus a page always has at least two lines.

During a pg session the window size can be changed with the w command (see **Miscellaneous commands**).

–number not specified:

The default window size is 23 lines.

–p *string*

Changes the prompt (or prompt string) for commands (see **COMMANDS DURING A PG SESSION**).

string

Is the new prompt string. If *string* includes the character sequence %d, the first occurrence of %d in *string* will be replaced by the current page number when pg issues the prompt string.

–p *string* not specified:

The default prompt is a colon (:).

option may be:

–c Clears the screen before displaying a new page.

–c not specified:

pg scrolls the previous page upward until the first line of the new page is positioned at the second line of the screen.

−e Causes pg not to pause at the end of each file; exits if no further files exist.

−e not specified:

The last line of a file is usually followed by (EOF) and the prompt.

Striking the <RETURN> key in response to the prompt causes pg to display the next file, or to terminate if there are no more files.

−f Inhibits pg from splitting lines longer than the screen width.

If the splitting of overlength lines is dictated by the hardware, the −f option cannot be used to override this feature. pg will count the lines as if no splitting occurred; however, the start of the affected page will no longer be visible on the screen.

−f not specified:

pg splits lines that extend beyond the right edge of the screen. Each line on the screen is thus treated as a separate line.

−n Automatic newline; causes pg to execute a command as soon as you enter the command letter in response to the prompt (see **COMMANDS DURING A PG SESSION**). With commands which do not include a letter, you press the space bar or <RETURN>.

−n not specified:

Every command must be terminated with <RETURN>.

−r Restricted mode. Shell commands are not permitted. If a shell command is entered, pg issues an error message but does not terminate.

−s Displays all messages and prompts in "standout" mode, which is usually inverse video.

+number

Starts displaying the file at a specified line number.

number

Line number at which the output is to begin.

+/RE/

Starts the output at the first line containing a string matching regular expression *RE*. *RE* is any simple regular expression (see expressions(5) , "Regular expressions").

This option is not effective unless the file is more than one page long (see *−number* above).

You can also search for strings interactively during a pg session (see **Pattern search commands**).

−− End of the list of options. Must be specified if *file* begins with −.

file_name

Name of the file to be paged. You can name more than one file, in which case pg displays the following before each file:

```
(Next file: file_name).
```

If you use a dash (-) as *file_name*, pg reads from standard input.

The commands n and p can be used to switch between various files during a session with pg (see **Miscellaneous commands**).

file_name not specified:

pg reads from standard input (see **COMMANDS DURING A PG SESSION**). Commands to pg are entered after the prompt on the last line and must normally be terminated by pressing the <RETURN>. You may, however, set the −n option and have them executed immediately.

During a session, pg may be in one of two states: displaying a page or waiting for the input of a command. The effect of pressing <DEL> or <CTRL-\> depends on the state pg is in:

- pg exits if it was in command mode, i.e. waiting for the input of a command.

- In visual mode, pg interrupts the display and issues its prompt, allowing you to enter new commands.

If you are using pg as a command in a pipeline, striking <DEL> or <CTRL-\> will affect other commands in the pipeline as well.

Let us assume that a grep command is writing to a pipe from which pg is reading input. If you press <DEL> while pg is displaying a page, grep will terminate, not pg; all data sent to the pipe by grep prior to its termination will still be read by pg.

Paging commands

<RETURN>
<SPACE>
On its own, scrolls to the next page.

number
pg scrolls to the page addressed by *number*, relative to the beginning of the file. Thus it makes no difference whether the specified page comes before or after the current page.

+[*number*]
Causes pg to scroll forward relative to the current page.

number
Number of pages to move forward; the farthest you can page forward is the last page of the file.

number not specified:

pg scrolls forward one page.

–[*number*]
Causes pg to scroll backward relative to the current page.

number
Number of pages to be scrolled back. The farthest you can page back is the first page of the file.

number not specified:

pg scrolls back one page.

[*number*]l
Causes pg to interpret the specified number as an absolute line address.

number
pg scrolls to the page addressed by *number*, relative to the beginning of the file. It makes no difference whether this line comes before or after the current line.

number not specified:

pg scrolls forward by one line.

+[*number*]l
Causes pg to scroll forward by the specified number of lines.

number
Number of lines by which pg is to scroll forward.

number not specified:

pg scrolls forward one line.

–[*number*]l
Causes pg to scroll back by the specified number of lines.

number
Number of lines by which pg is to scroll backward.

number not specified:

pg scrolls back one line.

[+]d
[+]<CTRL-D>
Scrolls half a screen forward. Use of the preceding plus sign + is optional.

−d
−<CTRL-D>
Scrolls half a screen back.

*number*f
Skips *number* screens of text.

*number*z
Same as <RETURN>. If specified, *number* becomes the new default number of lines per screenful.

.
<CTRL-L>
Causes the current page to be redisplayed.

$ pg displays the last page of the current file. $ should be used with caution when the input is read from a pipe.

Pattern search commands

[*k*]/*RE*/[*place*]
Searches forward for strings that match the specified *RE*, displaying the location at which the string appears.

The search begins with the page that comes immediately after the current page. If no matching string is found, the search terminates at the end of the current file.

k Causes pg to search for the *k*th occurrence of a matching string.

k not specified:

pg starts the output with the first matching string found.

RE A regular expression that defines the string to be located. *RE* must be a simple regular expression (see expressions(5) , "Regular expressions").

place

Location at which the matching string is to be displayed on the screen. You can enter:

m to place the line containing the matching string in the middle of the screen;

b to place the line with the matching string at the bottom of the screen;

t to place the line with the matching string at the top of the screen.

place not specified:

The line with the matching string is placed at the top of the screen.

[*k*]?*RE*?[*place*]
[*k*]^*RE*^[*place*]
Searches backward for strings that match the specified *RE*, displaying the location at which the string appears.

The search begins with the page that immediately precedes the current page. If no matching string is found, the search terminates at the beginning of the current file.

k Causes pg to search for the *k*th occurrence of a matching string.

k not specified:

pg starts the output with the first matching string found.

RE A regular expression that defines the string to be located. *RE* must be a simple internationalized regular expression (see expressions(5) , "Regular expressions").

place

Location at which the matching string is to be displayed on the screen. You can enter:

`m` to place the line containing the matching string in the middle of the screen;

`b` to place the line with the matching string at the bottom of the screen;

`t` to place the line with the matching string at the top of the screen.

place not specified:

The line with the matching string is placed at the top of the screen.

Miscellaneous commands

`h` Help function; displays a summary of available commands.

[*i*]`n` Goes to the *i*th next file named on the `pg` command line, making this file the current file. If there is not a sufficient number of files available, the last file becomes the current file.

i not specified:

The next file named on the command line becomes the current file.

[*i*]`p` Goes to the *i*th previous file named on the `pg` command line, making this file the current file. If there is not a sufficient number of files available, the first file becomes the current file.

i not specified:

The previous file named on the command line becomes the current file.

i`w` Sets the window size to *i* lines.

The default window size is 23 lines. You can also modify this setting when you call `pg` (see *–number* above).

`s`*file_name*

Saves the current file under the specified *file_name*.

`!`*command*

Passes the named *command* to the command interpreter named in the `SHELL` environment variable. If this variable is not defined, the default command interpreter is used.

This command must always be terminated with the <RETURN> key, even if you have invoked `pg` with the `–n` option.

`q`
`Q` Quits the `pg` command. You can also quit `pg` by pressing <RETURN> at the end of the last file named on the command line.

ENVIRONMENT VARIABLES

`COLUMNS`

Defines the number of columns for the screen. This value takes precedence over the number of columns defined for the `TERM` variable.

`LINES`

Defines the number of lines for the screen. This value takes precedence over the number of lines defined for the `TERM` variable.

`SHELL`

This environment variable determines the command interpreter to which `pg` passes shell commands entered during a `pg` session (see the `!` command).

`TERM` Type of terminal on which `pg` is executed.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_COLLATE` governs the collating sequence.

`LC_TYPE` governs character classes and character conversion (shifting).

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Locate the first occurrence of the word `example` in the file `text`:

```
$ pg +/example/ text
```

SEE ALSO

`cat(1)`, `ed(1)`, `grep(1)`, `more(1)`, `terminfo(4)`, `expressions(5)`.

plot(1)
BSD Compatibility Package

NAME

`plot, aedplot, bgplot, crtplot, dumbplot, gigiplot, hpplot, implot, t300, t300s, t4013, t450, tek -` graphics filters for various plotters

SYNOPSIS

`/usr/ucb/plot` [`–T`*terminal*]

DESCRIPTION

`plot` reads plotting instructions from the standard input and produces plotting instructions suitable for a particular *terminal* on standard output.

If no *terminal* is specified, the environment variable `TERM` is used. The default *terminal* is `tek`.

ENVIRONMENT

Except for `ver`, the following terminal-types can be used with "`lpr –g`" (see `lpr`) to produce plotted output:

`2648` | `2648a` | `h8` | `hp2648` | `hp2648a`
: Hewlett Packard 2648 graphics terminal.

`300` DASI 300 or GSI terminal (Diablo mechanism).

`300s` | `300S`
: DASI 300s terminal (Diablo mechanism).

`450` DASI Hyterm 450 terminal (Diablo mechanism).

`4013` Tektronix 4013 storage scope.

`4014` | `tek`
: Tektronix 4014 and 4015 storage scope with Enhanced Graphics Module. (Use 4013 for Tektronix 4014 or 4015 without the Enhanced Graphics Module).

`aed` AED 512 color graphics terminal.

`bgplot` | `bitgraph`
: BBN bitgraph graphics terminal.

`crt` Any crt terminal capable of running `vi(1)`.

`dumb` | `un` | `unknown`
: Dumb terminals without cursor addressing or line printers.

`gigi` | `vt125`
: DEC vt125 terminal.

`h7` | `hp7` | `hp7221`
: Hewlett Packard 7221 graphics terminal.

`implot`
: Imagen plotter.

`var` Benson Varian printer-plotter

`ver` Versatec D1200A printer-plotter. The output is scan-converted and suitable input to "`lpr –v`".

FILES

`/usr/ucb/aedplot`
`/usr/ucb/bgplot`
`/usr/ucb/crtplot`
`/usr/ucb/dumbplot`
`/usr/ucb/gigiplot`

/usr/ucb/hpplot
/usr/ucb/implot
/usr/ucb/plot
/usr/ucb/t300
/usr/ucb/t300s
/usr/ucb/t4013
/usr/ucb/t450
/usr/ucb/tek
/usr/ucb/vplot
/var/ucb/vplot*nnnnnn*

SEE ALSO

lpr(1), vi(1).

User's Reference Manual

pr(1)

NAME

`pr` - prepare files for printing

SYNOPSIS

`pr` [*option ...*] [*file ...*]

DESCRIPTION

The `pr` command formats and writes the contents of files on the standard output.

Output is either in single columns (default) or in multiple columns. Multi-column output can be produced by using either the – option or the `–m` option.

OPTIONS

No option specified:

The files are divided into pages which are separated by a sequence of line feed characters. The page length is 66 lines, which includes 10 lines of header and trailer output.

Headers are made up of two blank lines, one line of text containing the page number, date and time that the file was last modified, and the file name, and two more blank lines. Trailers consist of five blank lines.

Files are output in single columns. Lines that are too long are truncated.

If the standard output is associated with a terminal, error messages are not reported until all the specified files have been output.

The following overview indicates the various options you can use to modify the output format:

Function	Option
Set first page to be displayed	`+`
Split text into columns	`–`
Define how columns are to be filled	`–a`
Define page width for multi-column output	`–w`
Fold longer input lines	`–W`
Separate text columns	`–s`
Output files simultaneously in columns	`–m`
Double-space the output	`–d`
Convert tabs to spaces	`–e`
Convert spaces to tabs	`–i`
Number lines	`–n`
Indent text	`–o`
Change page length	`–l`
Change file name in header	`–h`
Display file one page at a time	`–p`
Use form-feed characters to separate pages	`–f`

Function	Option
Use form-feed characters to separate pages	–F
Suppress error messages	–r
Suppress output of header and trailer	–t

Description of options in alphabetical order

+*page_number*

Starts output at the specified *page_number*.

+ not specified: Output begins at the first page.

–*columns*

Outputs the file in *columns* columns. Output appears as if –e and –i have been set with their default values.

This option cannot be combined with –m.

The columns of a page are filled with input lines from top to bottom.

This setting can be changed with –a.

The default page width for multi-column output is 72. This setting can be changed with –w.

pr determines the width of each column by dividing the page width by the number of columns. If a line is too long, it is truncated to the right. Truncation can be suppressed with –s.

– not specified: Single-column output.

Warning:

Form feeds at the beginning of a line of text (^L or **TH0** L = Hex Code 0C) are ignored in multi-column output.

–a Arranges multi-column output across the page.

The number of columns must be greater than 1. This number can be defined with –*columns* or –w.

If a line is too long to fit in a column, it is truncated to the right. Truncation can be suppressed with –s.

The –a option cannot be combined with the –m option.

–a not specified:

pr fills columns down the page from top to bottom.

–d Double-spaces the output. Blank lines that result from double-spacing are deleted when they occur at the top of a page.

–e[*tab_char*][*spacing*]

Replaces each tab character in the input by an appropriate number of spaces.

tab_char

Character which pr interprets as a tab character. May be any non-numeric character.

tab_char not specified:

pr uses the horizontal tab character [see ascii(5)].

spacing

Spacing between tabs. The first tab stop in a line is always set in column 1. If *spacing* is 0, the default value of 8 characters is assumed.

spacing not specified: The tab spacing is 8 characters.

–f Uses a single form-feed character to separate output pages [see ascii(5)].

If the output is to a terminal, pr pauses before the first page and sounds the terminal bell. You then start the output by pressing RETURN.

–f not specified:

Pages are separated by a sequence of line feed characters.

`-F` Pages are separated by a single form feed character.

`-F` not specified:

The pages are separated by a sequence of line feed characters.

`-h` *header*

Writes a text of your choice in the header instead of the file name. This option is ignored if you also specify `-t` or `-l` with a page length of 10 or less.

`-i`[*tab_char*][*spacing*]

Replaces white space in the input text with the tab character *tab_char* at appropriate positions in the output text.

tab_char

Character which `pr` interprets as a tab character. May be any non-numeric character.

tab_char not specified:

`pr` uses the horizontal tab character [see `ascii(5)`].

spacing

Spacing between tabs. The first tab stop in a line is always set in column 1. If *spacing* is 0, the default value of 8 characters is assumed.

spacing not specified: The tab spacing is 8 characters.

`-l` *length*

Sets the length of an output page.

The page length includes a total of 10 lines for the header and footer. Consequently, if you specify a value of 10 or less for *length*, the header and trailer will not be output (see the `-t` option).

`-l` not specified: Each page is made up of 66 lines.

`-m` Merges and outputs all named files simultaneously, one column per file. The maximum number of files that may be specified is nine.

`-m` cannot be used together with *-columns*.

Otherwise, the same rules apply as for *-columns*.

The `-m` option cannot be combined with the `-a` option.

`-m` not specified: `pr` outputs files one after the other.

`-n`[*separator*][*char_pos*]

Numbers the lines. For multi-column output, the lines of each individual column are numbered. The line number occupies the first

char_pos+1 character positions of each line or each column line.

separator

Character used to separate the line number from the start of the line. May be any non-numeric character.

separator not specified:

A horizontal tab is used as the separator [see `ascii(5)`].

char_pos

Number of character positions occupied by line numbers.

char_pos not specified: Defaults to 5.

`-o` *offset*

Offsets (indents) each output line by *offset* character positions.

`-p` If the output is directed to a terminal, `pr` sounds the terminal bell and pauses before beginning each page. The page is displayed after you press the RETURN key.

`-r` Suppresses diagnostic reports if `pr` is unable to open a file.

`-r` not specified:

After completing the entire output, `pr` issues an error message to report files that could not be accessed.

`-s`[*separator*]

Separates columns by the single character *separator* instead of a tab. If you do not specify `-w` at the same time, `-s` also prevents the truncation of overlength lines (up to 512 characters) on multi-column output.

separator

Character used to separate columns.

separator not specified:

The default separator is a horizontal tab [see `ascii(5)`].

`-t` Suppresses the headers and trailers. Terminates printing after the last line of each file instead of padding with blanks to the end of the page.

`-w` *width*

Sets the page width for multi-column output. Multi-column output can be produced with *-columns* or `-m`.

width Number of characters in a line.

`-w` not specified:

The default page width for multi-column output is 72 characters.

`-W` Fold the lines of the input file. When used in multi-column mode (with the `-a` or `-m` options) lines will be folded to fit the current column's width, otherwise they will be folded to fit the current line width (80 columns).

`--` If *file* begins with a dash (`-`), the end of the command line options must be marked with `--`.

file Name of the file to be prepared for printing. You may specify more than one file.

If more than one file is specified, `pr` outputs them in succession.

If you use a dash as the name for *file*, `pr` reads from standard input.

If you use the `-m` option for multi-column output, no more than 9 files may be specified.

file not specified: `pr` reads from standard input.

EXAMPLES

Example 1

Print `file1` and `file2` sequentially in three-column format (`-3`) using the form-feed character (`-f`) at the end of each page so as to ensure that the output is independent of the printer type:

```
$ pr -3f file1 file2 | lp
```

Example 2

The tab spacing of 8 positions in `file1` is to be changed to 6 in `file2`. First you convert tabs in `file1` to spaces (`-e`; tab spacing = 8), then you reconvert the spaces back to tabs (`-i6`; tab spacing = 6), and then you write the result to `file2`. The `-t` option suppresses the header and trailer for both `pr` commands:

```
$ pr -et file1 | pr -i6t > file2
```

Example 3

The file `months` contains the name of a month in each line. This file is now to be displayed with `pr` in three columns (`-3`) with two-digit numbering (`-n2`). The columns are to be filled from left to right (`-a`):

```
$ pr -3n2a months
Jun 17 11:09 1991 months Page 1
 1      January          2      February         3      March
 4      April            5      May              6      June
 7      July             8      August           9      September
```

```
10      October         11      November        12      December
```

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

LC_TIME governs the format of date and time strings in the headers.

LC_TYPE specifies which characters are viewed as non-printing characters. Although non- printing characters appear in the standard output, they are not included in the calculation of line length and column width.

If LC_MESSAGES, LC_TIME or LC_TYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

FILES

/dev/tty*
/dev/term/tty*
: Special files for individual terminals.

  If standard output is directed to one of the special files /dev/tty*, other output directed to this terminal is delayed until standard output is completed. This prevents error messages from being interspersed throughout the output.

SEE ALSO

cat(1), expand(1), fmt(1), fold(1), lp(1), more(1), page(1), pg(1), ascii(5).

User's Reference Manual

printenv(1)
BSD Compatibility Package

NAME

`printenv` - display environment variables currently set

SYNOPSIS

`/usr/ucb/printenv` [*variable*]

DESCRIPTION

`printenv` prints out the values of the variables in the environment. If a *variable* is specified, only its value is printed.

DIAGNOSTICS

If a *variable* is specified and it is not defined in the environment, `printenv` returns an exit status of `1`.

SEE ALSO

`csh`(1), `echo`(1), `sh`(1), `stty`(1), `tset`(1), `environ`(5).

User's Reference Manual

printf(1)

NAME

`printf` - formatted output

SYNOPSIS

`printf` '*format*'[*arg* ...]

DESCRIPTION

The `printf` command outputs the arguments you specify in formatted form. `printf` supports most of the format specifications for strings as in the `printf(3S)` function in C.

OPERANDS

'*format*'

Character string that can contain three different types of objects:

- plain characters, which are output without any modifications.
- C-language escape sequences, which are converted into the corresponding characters in the output, e.g. `\n` is converted to a newline character.
- Format elements from which each one of the specified arguments *arg* is processed.

Metacharacters

The following metacharacters are interpreted by `printf`:

`\\` Backslash (for distinguishing octal characters)

`\a` Warning, alert

`\b` Backspace

`\f` Form feed

`\n` Newline

`\r` Carriage Return

`\t` Tab

`\v` Vertical tab

`\`*octal*

Octal number, whereby *octal* consists of one, two, or three digits

Format elements

A conversion specification comprises:

`%`[*arg_no*`$`][*field_width*][`.`*precision*]*conversion_character*

`%` Always located at the beginning of the conversion specification. If the `%` character is not to be interpreted as a part of the conversion specification but as an ordinary character to be output, it must be escaped by preceding it with another `%` (`%%`).

arg_no`$`

Decimal integer with which you specify the position of the argument to be processed. The number must be followed by a `$` character.

arg_no not specified: The argument following the last converted argument is used.

Note: If you use *arg_no*`$` for one argument, you should also use *arg_no*`$` for all the other arguments.

field_width

Decimal integer with which you specify the minimum field width. If the string or number to be converted has fewer characters/digits than *field_width*, it is padded on the left to the field width

and output right-adjusted. If left-adjustment is desired, the decimal integer must be preceded by a dash (–). The padding is with blanks unless the *field_width* integer starts with a zero, in which case padding for right-adjusted output is done with zeros.

A *field_width* may also be indicated by an asterisk (*) instead of an integer. In this case, an integer argument supplies the field width. This argument must appear before the string or number to be converted. The asterisk does not work in combination with *arg_no*$.

If the string or number is longer than the field width, the field is automatically expanded.

.precision

Decimal integer with which you specify the maximum number of characters to be output from the string or number to be converted. This number must be preceded by a dot (.). If the precision argument is zero, nothing is output. The number of characters output is always controlled by the precision, even if some other value has been specified for the field width.

A *precision* may also be indicated by an asterisk (*) instead of an integer. In this case, an integer argument supplies the precision. This argument must appear before the string or number to be converted. The asterisk does not work in combination with *arg_no*$.

conversion_character

The following *conversion_characters* can be used for `printf`: .ift\{\

Conversion character	Meaning
`b`	character string with metacharacters
`c`	single character
`d`, `i`	signed decimal integer
`e`, `E`	floating point number in exponential notation, e.g. 5.234e+2
`f`	floating point number, e.g. 52.34
`g`, `G`	`%e` or `%f`, whichever is shorter
`o`	signed octal integer (base 8)
`s`	character string
`u`	unsigned decimal integer
`x`, `X`, `p`	unsigned hexadecimal integer (base 16)

With `s`, all characters from the character string are output until the number of characters specified for *precision* is reached. If *precision* is not specified, the entire character string is output.

With `b`, the character string in *arg* can contain metacharacters. `printf` supports all escape sequences interpreted by the `echo` command in this case, i.e. the escape sequences specified above with the exception of octal numbers specified as `\0`*octal*, and `\c`. `\c` causes `printf` to abort output at this point and not to terminate it with a newline character.

arg String to be written to standard output in the format specified by *format*. If there are less arguments available than are required by *format*, 0 or the null string are used for the missing arguments. If more arguments are available than are required by *format*, *format* is used several times (except where *arg_no*$ is specified; the excess arguments are ignored in this case).

arg not specified:

The result is undefined.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If

LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Output the string "Good Morning Helen" on the screen:

```
$ printf '%s %s %s\n'Good Morning Helen
```

The following command produces the same output:

```
$ printf '%2$s %3$s %1$s\n'Helen Good Morning
```

Example 2

Output the first 6 characters of your home directory /usr/kathy with an appropriate message:

```
$ printf 'The first 6 characters of %s are %.6s.\n' $HOME $HOME
The first 6 characters of /usr/kathy are /usr/k.
```

Example 3

The examples that follow write a 4-digit number in an 8-character field right-adjusted, right-adjusted with leading zeros, and left-adjusted, respectively:

```
$ printf '%8s\n' 1860
    1860
$ printf '%08s\n' 1860
00001860
$ printf '%-8s\n' 1860
1860ERRORERRORERRORERROR
```

Example 4

Output a fixed point number limited to two decimal places:

```
$ printf '%.2f\n' 1000.3333333
1000.33
```

Note: Where the LANG variable is German, a decimal *comma* must be entered instead of the decimal point.

SEE ALSO

awk(1), bc(1), echo(1), printf(3S).

User's Reference Manual

priocntl(1)

NAME

`priocntl -` set scheduling parameters and priorities (prioritycontrol)

SYNOPSIS

Format 1

`priocntl –l`

Format 2

`priocntl –d` [`–i` *type*] [*list*]

Format 3

`priocntl –s` [`–c` *class*] [*options ...*] [`–i` *type*] [*list*]

Format 4

`priocntl –e` [`–c` *class*] [*options ...*] *command* [*arguments*]

DESCRIPTION

The `priocntl` command can be used to

- display or set scheduling parameters (i.e. the priorities and time quantum allotments) of the specified processes.
- display the current configuration information for the system's process scheduler. The scheduler is the process that controls the scheduling parameters for all processes.
- execute a command with specific scheduling parameters.

Processes fall into distinct classes with a separate scheduling policy applied to each class. The three process classes currently supported are the `real-time` class, the `time-sharing` class and the `fixed-priority` class. The characteristics of these classes and the class-specific options they accept are described in the sections entitled **Real-time class**, **Time-sharing class**, and **Fixed-priority class**. Provided you have the appropriate permissions, you can use `priocntl` to change the class and other scheduling parameters associated with a running process.

Caution:

A runnable real-time process runs before any other process. For this reason, inappropriate use of real-time processes can have a dramatic negative impact on system performance.

Format 1: Display information on all classes

`priocntl –l`

`–l` Displays all the classes currently configured on the system together with class-specific information about each class. The format and meaning of this information is described in the appropriate section below.

Format 2: Display class and parameters for processes

`priocntl –d` [`–i` *type*] [*list*]

`–d` Displays the class and class-specific scheduling parameters for the specified processes.

`–i` *type*

The `–i` option and its associated *type* argument, together with the *list* arguments (if any), specify one or more processes to which the `priocntl` command is to apply. How *list* is interpreted depends on the *type* specified. Valid *type* values and the corresponding interpretations of *list* are as follows:

`–i pid`

list is a list of process IDs. The `priocntl` command applies to all processes whose process IDs are in the list.

`-i ppid`
list is a list of parent process IDs. The `priocntl` command applies to all processes whose parent process IDs are in the list.

`-i pgid`
list is a list of process group IDs. The `priocntl` command applies to all processes that belong to a process group in the list.

`-i sid`
list is a list of session IDs. The `priocntl` command applies to all processes that belong to one of the sessions in the list.

`-i class`
list is the name of a single class, which you may specify as `RT` for real-time, `TS` for time-sharing, or `FP` for fixed-priority. The `priocntl` command applies to all processes in the specified class.

`-i uid`
list is a list of user IDs. The `priocntl` command applies to all processes whose effective user IDs are in the list.

`-i gid`
list is a list of group IDs. The `priocntl` command applies to all processes whose effective group IDs are in the list.

`-i all`
The `priocntl` command applies to all existing processes. *list* is ignored in this case. However, the list of processes may be restricted if you do not have the required permissions.

`-i` *type* not specified:

`pid` is used as the default identification type.

list *list* specifies the processes to which the `priocntl` command is to apply. The meanings of the elements in the list are described under the `-i` option. If a *list* is specified, it must be at the end of the command line. The individual elements in the list must be separated by blanks or tabs.

list not specified:

If a *type* argument of `pid`, `ppid`, `pgid`, `sid`, `class`, `uid`, or `gid` is specified with the `-i` option, the process ID, parent process ID, process group ID, session ID, class, user ID, or group ID respectively of the `priocntl` command itself is used.

Format 3: Set class and parameters for processes

`priocntl -s` [`-c` *class*] [*options* ...] [`-i` *type*] [*list*]

`-s` Sets the class and class-specific parameters of the specified processes to the values given on the command line.

`-c` *class*
class specifies the class to be set for scheduling. Valid *class* arguments are `RT` for real-time, `TS` for time-sharing, and `FP` for fixed-priority.

`-c` *class* not specified:

type and *list* must specify a set of processes which are all in the same class. Otherwise, `priocntl` will abort with an error message.

options
Specifies the class-specific options. Valid options for these classes are described in the sections entitled **Real-time class**, **Time-sharing class**, and **Fixed-priority class**.

No options specified:

The class-specific parameters are set to the default values for the class specified by `-c` *class*, or to the default values for the process's current class if the `-c` *class* option has also been omitted.

`-i` *type*

This option specifies the processes whose parameters are to be set. Possible values are described under **Format 2**.

list Together with –i *type*, *list* specifies the processes whose parameters are to be set. See **Format 2** for more details.

In order to change the scheduling parameters of a specific process with priocntl, your real or effective user ID must match the real or effective user ID of this process, unless you have the system administrator's effective user ID. These are the minimum requirements enforced for all classes. An individual class may impose additional restrictions when setting processes to that class or when setting class-specific parameters.

When –i *type* and *list* specify a list of processes, priocntl acts on these processes in implementation-specific order. If priocntl encounters an error for one or more of the specified processes, it may or may not continue through the list of processes, depending on the nature of the error. If the error is due to insufficient permissions, priocntl issues an error message and then continues through the rest of the processes, resetting the parameters for all processes for which you have the required permissions. If the cause of the error is not due to permissions, priocntl terminates immediately without acting on the other processes in the list and issues an error message.

A special sys scheduling class exists for scheduling the execution of certain system processes (such as the swapper process, which swaps data in and out of disk storage). It is not possible to change the class of any process to sys. In addition, any processes in the sys class are ignored if they are specified with –i *type* and *list*.

Example:

If *type* were uid, a *list* consisting of a zero would specify all processes with a user ID of zero except processes in the sys class and (if changing the parameters using the –s option) the init process.

The init process (process ID 1) is a special case. If you want priocntl to change the class or other scheduling parameters of the init process, you must specify pid for *type* and 1 for *list*. You may assign the init process to any class configured on the system, but the time-sharing class is almost always the appropriate choice. Other classes may have negative effects.

Format 4: Execute a command with specific parameters

priocntl –e [–c *class*] [*option ...*] *command* [*arguments*]

–e Executes *command* with the class and scheduling options specified.

–c *class*

class specifies the class to be set for the command to be executed. Valid *class* arguments are RT for real-time, TS for time-sharing, and FP for fixed-priority.

–c *class* not specified:

The command is executed in the user's current class.

option

Specifies the class-specific options. Valid options for both classes are described in the sections entitled **Real-time class**, **Time-sharing class**, and **Fixed-priority class**.

No options specified:

The command is executed with the default values for the class specified by –c *class*, or with the default values for the user's current class if the –c *class* option has also been omitted.

command

The command to be executed with the specified parameters in the specified class.

arguments

Arguments for the command to be executed.

Real-time class

The real-time class provides fixed-priority preemptive CPU scheduling for those processes requiring fast and deterministic response and absolute user control of scheduling priorities. If the real-time class is configured on

the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable real-time process is given CPU service before any other process.

A real-time priority (*rtpri*) may be assigned to a process in the real-time class. Real-time priorities range from 0 to *x*, where the value of *x* is configurable and can be displayed for a specific configuration by means of the `priocntl -l` command.

Real-time scheduling is fixed-priority scheduling. The priority of a real-time process never changes unless the user or an appropriate program changes it explicitly.

For processes in the real-time class, the *rtpri* value is, for all practical purposes, equivalent to the scheduling priority of the process. The *rtpri* value determines the scheduling priority of a real-time process relative to other processes within its class. A higher *rtpri* value represents a higher priority. Since the real-time class controls the highest range of CPU scheduling priorities in the system, it is guaranteed that a runnable process with the highest *rtpri* value is always assigned CPU time before any other process in the system.

In addition to using `priocntl` to control priorities, you can also use it to influence the length of the time quantum allotted to a process in the real-time class. The time quantum value specifies the maximum amount of CPU time a process is assigned, provided that it does not enter an external event wait state (`sleep`) such as during hard disk access. If another process becomes runnable with a higher priority, the currently running process may be preempted before receiving its full time quantum.

The command

`priocntl -d`[`-i` *type*] [*list*]

can be used to display the priority and time quantum (in milliseconds) for each process in the real-time class.

Valid options for setting real-time parameters (`-s` and `-e`) are:

`-p` *rtpri*

Sets the real-time priority of the specified process(es) to *rtpri*.

Option `-p` not specified:

If the process is already in the real-time class, the priority remains unchanged. If the class of a process is changed to real-time from some other class, the priority defaults to a value of 0.

`-t` *tqntm*

Sets the time quantum for the specified process(es) to *tqntm*. If 0 is specified or a value greater than the maximum permitted, `priocntl` aborts with an error message. The maximum value depends on the implementation.

Option `-t` not specified:

If the process is already in the real-time class, the time quantum remains unchanged. If the class of a process is changed to real-time from some other class, the time quantum defaults to a value dependent on the priority and system configuration [see `rt_dptbl(4)`].

`-r` *resolution*

resolution defines the units in which the time quantum is specified with the `-t` option. *resolution* is a positive integer between 1 and 1,000,000,000 inclusive. The resolution used is the reciprocal of *resolution* in seconds.

Example:

`-t 10 -r 100` would set the resolution to hundredths of a second and the resulting time quantum length would be 10/100 seconds (one tenth of a second).

Although very high (nanosecond) resolution may be specified, the specified time quantum length is always rounded up by the system to the next integral multiple of the system clock's resolution.

Example:

The highest resolution currently available on a system is 10 milliseconds. If the `-t` and `-r` options are used to specify a time quantum of 34 milliseconds, it will be rounded up to 40 milliseconds on that system.

The `-r` option can only be specified together with the `-t` option.

Option –r not specified:

1 millisecond resolution is assumed for the time quantum set with –t.

Any combination of the –p and –t options (and possibly –r) may be used together with –s or -e.

Only the system administrator may change the class of a process from any other class to real-time. To change the priority or the time quantum of a real-time process, you must either have system administrator privileges, or you must currently be in the real-time class (i.e. your command interpreter must be running as a real-time process) and have the same real or effective user ID as the process whose parameters are to be changed.

The real-time priority and time quantum are inherited across the fork(2) and exec(2) system calls.

Time-sharing class

The time-sharing scheduling policy provides for a fair and effective allocation of CPU time among processes with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive processes and good throughput to CPU-bound jobs while providing a degree of user control over scheduling.

A time-sharing user priority (tsupri) may be assigned to a process in the time-sharing class. Time-sharing priorities range from –*x* to +*x*, where the value of *x* is configurable and can be displayed by for a specific configuration by means of the priocntl –l command.

The purpose of the time-sharing priority is to provide some degree of user control over the scheduling of processes in the time-sharing class. Raising or lowering the *tsupri* value of a process in the time-sharing class raises or lowers the scheduling priority of the process. It is not guaranteed, however, that a time-sharing process with a higher *tsupri* value will run before one with a lower *tsupri* value. This is because the *tsupri* value is just one factor used to determine the scheduling priority of a time-sharing process. The system may dynamically adjust the internal scheduling priority of a time-sharing process based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (displayed with priocntl –l), there is a per-process user priority limit (time-sharing user priority limit, *tsuprilim*).

The command

priocntl –d [–i *type*] [*list*]

can be used to display the user priority and user priority limit for each specified time-sharing process.

Valid options for setting time-sharing parameters (–s and –e) are:

–m *tsuprilim*

May only be set in conjunction with –e or –s. Sets the user priority limit of the specified process(es) to *tsuprilim*.

Any time-sharing process may lower its own user priority limit or that of another process with the same user ID, but only a time-sharing process with system administrator privileges may raise the limit. When changing the class of a process to time-sharing from some other class, system administrator privileges are required in order to set a limit to a value greater than zero.

Option –m not specified:

If the process is already in the time-sharing class, the user priority limit remains unchanged. If the class of a process is changed to time-sharing from some other class, the limit defaults to a value of 0.

–p *tsupri*

May only be set in conjunction with –e or –s. Sets the user priority of the specified process(es) to tsupri.

Any time-sharing process may set its own user priority or that of another process with the same user ID to any value less than or equal to the process's priority limit. Attempts to set the tsupri above the *tsuprilim* or set the *tsuprilim* below the tsupri result in the tsupri being set equal to the *tsuprilim*.

Option –p not specified:

If the process is already in the time-sharing class, the user priority normally remains unchanged. If the class of a process is changed to time-sharing from some other class, the user priority defaults to the

value of the user priority limit set.

Exception:

If the –p option is not specified and –m is used to set a *tsuprilim* below the current *tsupri*, the *tsupri* is changed to the *tsuprilim* which is being set.

Any combination of the –m and –p options may be used with –s or –e.

The user priority and user priority limit are inherited across the fork(2) and exec(2) system calls.

Fixed-priority class

The fixed-priority class provides a fixed-priority preemptive scheduling policy for those processes requiring fast and deterministic response and absolute user/application control of scheduling priorities.

The fixed-priority class has a range of fixed-priority priority (*fppri*) values that may be assigned to processes within the class. Fixed-priority priorities range from 0 to *x*, where the value of *x* is configurable and can be displayed for a specific installation by using the command priocntl –l.

As the name implies, the fixed-priority scheduling policy is a fixed-priority policy. The scheduling priority of a fixed-priority process never changes except as the result of an explicit request by the user/application to change the *fppri* value of the process.

For processes in the fixed-priority class, the *fppri* value is, for all practical purposes, equivalent to the scheduling priority of the process. The *fppri* value completely determines the scheduling priority of a fixed-priority process relative to other processes within its class. Numerically higher *fppri* values represent higher priorities.

In addition to providing control over priority, priocntl provides for control over the length of the time quantum allotted to processes in the fixed-priority class. The time quantum value specifies the maximum amount of time a process may run assuming that it does not complete or enter a resource or event wait state (sleep). Note that if another process becomes runnable at a higher priority the currently running process may be preempted before receiving its full time quantum.

The

priocntl –d [–i *type*] [*list*]

command displays the fixed-priority priority and time quantum (in millisecond resolution) for each fixed-priority process in the set specified by *type* and *list*.

The valid class-specific options for setting fixed-priority parameters are:

–p *fppri*
: Set the fixed-priority priority of the specified process(es) to *fppri*.

–t *tqntm* [–r *resolution*]
: Set the time quantum of the specified process(es) to *tqntm*. You may optionally specify a resolution as explained below.

Any combination of the –p and –t options may be used with priocntl –s or priocntl –e for the fixed-priority class. If an option is omitted and the process is currently fixed-priority the associated parameter is unaffected. If an option is omitted when changing the class of a process to fixed-priority from some other class, the associated parameter is set to a default value. The default value for *rtpri* is 0 and the default for time quantum is dependent on the value of *fppri* and on the system configuration [see fp_dptbl(4)].

When using the –t *tqntm* option you may optionally specify a resolution using the –r *resolution* option. (If no resolution is specified, millisecond resolution is assumed.) If *resolution* is specified it must be a positive integer between 1 and 1,000,000,000 inclusive and the resolution used is the reciprocal of *resolution* in seconds. For example, specifying –t 10 –r 100 would set the resolution to hundredths of a second and the resulting time quantum length would be 10/100 seconds (one tenth of a second). Although very fine (nanosecond) resolution may be specified, the time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. Requests for time quantums of zero or quantums greater than the (typically very large) implementation-specific maximum quantum result in an error.

In order to set the time quantum to an infinite value, enter the INFINITE string instead of a value for *duration*.

If *duration* is set to the `DEFAULT` value, the time quantum value of the parameter table of the fixed-priority scheduler is used [see `fp_dptbl(4)`].

In order to change the class of a process to fixed-priority (from any other class) the user invoking `priocntl` must have super-user privilege. In order to change the *fppri* value or time quantum of a fixed-priority process the user invoking `priocntl` must either be super-user, or must currently be in the fixed-priority class (shell running as a fixed-priority process) with a real or effective user ID matching the real or effective user ID of the target process.

The fixed-priority priority and time quantum are inherited across the `fork(2)` and `exec(2)` system calls.

PROCESSOR AFFINITY AND BINDING CONSIDERATIONS

In order to increase the cache hit rates of a multiprocessor, a process should generally execute on a particular processor. This affinity for a particular processor will reduce the required memory bandwidth for the process (and for the system as a whole). If a process has not run "recently" then it probably only has a small cache state in any given processor – and thus should loose its affinity for any particular processor.

This affinity mechanism can be enabled or disabled on a per-process basis by the super user [see `mpcntl(3X)`]. The affinity mechanism defaults to `on` for the `TS` and `FP` classes and to `off` for the `RT` class.

A process can also be bound to a set of processors. If a processor has been so bound, then it will *only* run on one of the processors of that set. If there is a idle processor that is not in the processes binding set – then the process will still wait until there is a processor in its set that is not running a higher priority process.

Process bindings can be established by the process [see `mpcntl(3X)`], or can be temporarily established by the kernel (if the kernel is executing driver code that must run on a particular CPU).

ERROR MESSAGES

`Process(es) not found`
: None of the specified processes exists.

`Specified processes from different classes`
: The `–s` option was specified to set parameters; the `–c class` option was not specified; and the specified processes are from different classes.

`Invalid option or argument`
: An option or option argument cannot be recognized or is invalid.

`Permissions error encountered on pid` *x*
: You do not have the appropriate permissions to change the settings for the process with process ID *x*.

EXAMPLES

Example 1

The following command sets the class of any non-real-time processes identified by the specified process IDs to real-time and sets their real-time priority to the default value of 0. The real-time priorities of processes already in the real-time class remain unchanged. The time quantums of all of the specified processes are set to 1/10 seconds (1 unit with a resolution of 1/10 seconds).

```
$ priocntl –s –c RT –t 1 –r 10 –i pid 1234 2345 3456
```

Example 2

The next command executes the specified command in the real-time class with a real-time priority of 15 and a time quantum of 20 milliseconds.

`$ priocntl –e –c RT –p 15 –t 20` *command*

Example 3

The following command sets the class of all processes with UID 17 to time-sharing, assuming they are not already in that class. The user priority limit and the user priority are set to the default value 0. The time-sharing parameters of processes already in the time-sharing class remain unchanged:

```
$ priocntl –s –c TS –i uid 17
```

Example 4

In this example, the specified command is executed in the time-sharing class with a user priority limit of 0 and a user priority of –15:

`$ priocntl –e –c TS –m 0 –p –15` *command arguments*

Example 5

`priocntl` sets the class of any non-fixed-priority processes selected by *type* and *list* to fixed-priority and sets their fixed-priority priority to the default value of 0. The real-time priorities of any processes currently in the real-time class are unaffected. The time quantums of all of the specified processes are set to 1/10 seconds:

`priocntl –s –c FP –t 1 –r 10 –i` *type list*

Example 6

`priocntl` executes *command* in the fixed-priority class with a fixed-priority priority of 15 and a time quantum of 20 milliseconds:

`priocntl –e –c FP –p 15 –t 20` *command*

SEE ALSO

`nice(1)`, `ps(1)`, `priocntl(2)`, `fp_dptbl(4)`, `rt_dptbl(4)`, `ts_dptbl(4)`.

User's Reference Manual

ps(1)

NAME

`ps` - report process status

SYNOPSIS

`ps` [`-aAdeflcCjx`] [`-g` *grplist*] [`-G` *grplist*] [`-o` *format*] [`-p` *proclist*] [`-t` *termlist*] [`-u` *uidlist*] [`-U` *uidlist*] [`-s` *sesslist*]

DESCRIPTION

`ps` outputs information about active processes. This report represents a snapshot of the system or process status at a given moment and could be outdated within a split-second, so it may not reflect the actual situation by the time it is displayed.

OPTIONS

No option specified:

`ps` outputs information about processes associated with the current terminal and whose effective user ID corresponds to the user ID of the caller. The output consists of a short listing comprising:

- the process ID (`PID`)
- the terminal identifier (`TTY`)
- the cumulative execution time (`TIME`)
- the command name (`CMD`).

The significance of the output columns is explained in the section entitled **OUTPUT**.

`-a` Outputs information about all processes associated with a terminal, except process group leaders.

`-A` Outputs information about all processes. Similar to `-e`.

`-c` Also outputs information relating to process class and priority [see `priocntl(1)`].

`-C` Indicates whether or not the process is a clone process.

`-d` Outputs information about all processes associated with a terminal as well as those which are not. Process group leaders are not taken into account.

`-e` Outputs information about all processes. Similar to `-A`.

`-f` Outputs a full listing with supplementary information on each process. The columns displayed in a full listing are explained in more detail in the section entitled **OUTPUT**.

If `-f` is specified, `ps` outputs the command name and arguments. However, the arguments are displayed only if the process belongs to the user who called `ps`, or if `ps` was called by `root`.

If the command name for the process contains non-printing characters, the command name is enclosed within square brackets [...]. If the `-f` option is specified without other options, the information output refers to processes associated with the controlling terminal.

`-g` *grplist*

Restricts listing to data about processes whose process group leaders are given in *grplist*.

grplist

grplist is a list containing the process ID numbers of process group leaders. This list can be specified in one of two forms: a comma-separated list of numbers, or a list of numbers enclosed in double quotes with the numbers separated by commas and/or blanks.

`-G` *grplist*

same as `-g` with specification of the real group ID.

`-j` Outputs the session ID and the process group ID.

`-l` Outputs a long listing with detailed information on each process. The output columns in a long listing are explained in the section entitled **OUTPUT**. If the `-l` option is specified without other options, the information output refers to processes associated with the controlling terminal. The `-x` option can be specified for 64-bit systems (see also this option).

`-o` *format*
(o – output) Defines the output format. Several `-o` options can be specified. The specified formats are then output one after the other.

format
format is a list of variables and, where applicable, their headings for the individual output columns (see **OUTPUT**). *format* has the following format: The variables are separated by a comma, or *format* is enclosed in quotes "...", whereby the variables can then be separated by a comma and/or a blank. A new heading text must always appear at the end of the list. Additional variables are specified with a further `-o` option.

`-p` *proclist*
Restricts listing to data about processes whose process ID numbers are given in *proclist*.

proclist
proclist is a list of process ID numbers.

The numbers must be separated by commas, or alternatively, the whole list can be enclosed in double quotes with the numbers separated by commas and/or blanks.

`-s` *sesslist*
Restricts listing to data about processes associated with a session listed in *sesslist*.

sesslist
sesslist is a list of session ID numbers. This list can be specified in one of two forms: a comma-separated list of numbers, or a list of numbers enclosed in double quotes with the numbers separated by commas and/or blanks.

`-t` *termlist*
Restricts listing to data about the processes associated with the terminals named in *termlist*.

termlist
termlist is a list of terminal identifiers which may be specified in one of two forms: either the device's file name (e.g. `term/tty04`) or, if the device's file name is constructed with `tty`, just the digit identifier (e.g. `004`).

The terminal identifiers must be separated by commas, or alternatively, the whole list can be enclosed in double quotes with the entries separated by commas and/or blanks.

`-u` *uidlist*
Restricts listing to data about processes whose process owner is given in *uidlist*.

uidlist
uidlist is a list of user ID numbers or login names.

The entries in *uidlist* must be separated by commas, or alternatively, the whole list can be enclosed in double quotes with the entries separated by commas and/or blanks.

`-U` *uidlist*
same as `-u` with specification of the real user ID.

`-x` This option only works for 64-bit systems. The `ADDR` and `WCHAN` fields are represented as 64 bits long. The `-l` option must also be set for this option to be effective.

OUTPUT

Default output format

The headings and the meanings of the columns in a `ps` default listing are given below. The letters in

parentheses indicate the option that causes the corresponding column to appear. "all" means that the column appears for all options. Note that options `-c`, `-j`, `-f` and `-l` 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. These flags are machine-dependent and have therefore been omitted here.

`S` (`l`)
: State of the process:

  `O` Process running

  `S` Sleeping: process waiting for an event

  `R` Runnable: process is runnable

  `I` Idle: process being created

  `Z` Zombie state: process is terminated, but its exit status has not yet been queried by the parent with a `wait(2)` system call

  `T` Traced: traced process stopped by a parent process

  `X` SXBRK state: process waiting for more memory

`UID` (`f`, `l`)
: (= User ID) The user ID number of the process owner. If the `-f` option is set, the login name is output instead of the UID.

  Only the first 7 characters of the user ID are output.

`PID` (all)
: (= Process ID) The process ID number of the process. Every process is assigned a unique PID when it is created. You can use this number in a `kill` command, for example, if you want to terminate a particular process.

`PPID` (`C`, `f`, `l`)
: (= Parent Process ID) The process ID of the parent process.

`CLN` (`C`)
: Status of the clone process: 0 = no clone process, 1 = clone process.

`PGID` (`j`)
: Process group ID.

`SID` (`j`)
: Session ID.

`C` (`f`, `l`)
: Processor utilization for scheduling.

`CLS` (`c`)
: Scheduling class (processes handled by the scheduler).

`PRI` (`l`, `c`)
: Priority of the process. Higher numbers normally mean lower priority. However, if the `-c` option is specified, higher numbers mean higher priority.

`NI` (`l`)
: Nice value, i.e. value by which process priority was changed [see `nice(1)`]. Only processes in the *time-sharing* class have a nice value.

`ADDR` (`l`)
: Core address (physical page frame number) of the user area if resident; the disk address if swapped out.

`SZ` (`l`)
: Size in blocks of the core image of the process.

`WCHAN` (`l`)

Address of the event for which the process is waiting. If the column is blank, the process is running.

`STIME` (`f`)
Starting time of the process. The time is output within the first 24 hours; thereafter, the date.

`TTY` (all)
The controlling terminal for the process. A ? is output when there is no controlling terminal.

`TIME` (all)
The cumulative execution time for the process in minutes and seconds.

`CMD` (all)
The command name. If the `-f` option is specified, the full command name and its arguments are output.

Processes which have terminated, but whose exit status has not yet been queried by the parent with a `wait()` system call, is marked `<defunct>`.

If *termlist*, *proclist*, *uidlist*, or *grplist* is not specified, `ps` will try to determine the controlling terminal by checking the standard input, standard output, and standard error in that order. It will then report on the processes associated with the controlling terminal. If the above three channels are all redirected, `ps` will not find a controlling terminal and hence not produce a report.

User-defined output format

You can create your own output format using the `-o` option. Each column of the output format is represented by one of the variables listed below.

The heading assigned by default can be changed by entering an equals sign after the variable and then specifying the text for the heading.

Example: `-o user=`*user_name*

A new heading text must always appear at the end of the list of variables. Further variables can then be specified using the `-o` option.

The fields defined with the `-o` option are output in the specified sequence. The field lengths are at least as long as the defined heading text. If the new heading text is empty (e.g. `-o user=`), the field length corresponds to at least the default heading. If all of the heading texts are empty, no heading lines are output.

`ruser`
Real user number of the process owner. Where possible, the login name is output instead of the UID.

Default heading: `RUSER`

`user` Effective user number of the process owner. Where possible, the login name is output instead of the UID.

Default heading: `USER`

`rgroup`
Real ID of the process group. Where possible, the group name is is output instead of the GID.

Default heading: `RGROUP`

`group`
Effective ID of the process group. Where possible, the group name is output instead of the GID.

Default heading: `GROUP`

`pid` Process ID number (decimal).

Default heading: `PID`

`ppid` Process ID number of the parent process (decimal).

Default heading: `PPID`

`clone`
Status of the clone process (decimal).

Default heading: `CLN`

`pgid` ID number of the process group (decimal).

Default heading: `PGID`

`vsz` Size of the process in the virtual memory in Kbytes.

Default heading: `VSZ`

`nice` Priority value of the process (decimal) [see `nice(1)`].

Default heading: `NI`

`etime`

Time expired since the start of the process. The time is specified in the following format:

[[*dd*–]*hh*:]*mm*:*ss*

dd stands for the number of days, *hh* for the number of hours, *mm* for the number of minutes, and *ss* for the number of seconds.

Default heading: `ELAPSED`

`time` CPU time expired since the start of the process. The time is specified in the following format:

[*dd*–]*hh*:*mm*:*ss*

dd stands for the number of days, *hh* for the number of hours, *mm* for the number of minutes, and *ss* for the number of seconds.

Default heading: `TIME`

`tty` Data display terminal controlling the process.

Default heading: `TT`

`comm` Name of the executed command.

Default heading: `COMMAND`

`args` Name of the executed command with all arguments. The list of arguments can be truncated for space reasons.

Default heading: `COMMAND`

EXAMPLES

`ps` is called with the `–l` option to display a long listing of information on all active processes on the controlling terminal:

```
$ ps –l
 F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME CMD
10 S   110  1455  1453  0  30 20 c0fec9d8     23 d114f200 tty004  0:02 sh
10 O   110  1862  1455 12  50 20 c0fec870     60          tty004  0:00 ps
18 S   110  1858  1455 10  20 20 c0fecaf8     55 d115f280 tty004  1:03 find
```

The `find` command is now to be terminated by calling `kill` with the process ID number displayed in the PID column. A subsequent call to `ps` then confirms that the corresponding process no longer exists.

```
$ kill 1858
$ ps
   PID TTY      TIME COMD
  1455 tty004   0:02 sh
  1873 tty004   0:00 ps
1858 Terminated
```

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_TIME` governs the format of time strings when used in conjunction with the `–f` option.

`LC_COLLATE` governs the collating sequence.

If `LC_MESSAGES`, `LC_TIME` or `LC_COLLATE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

NOTES

If an I/O-intensive process is running, it may not be recognized by `ps` because the accompanying data is locked.

FILES

`/dev/sxt/*`
: Special files for virtual tty devices

`/dev/term/tty*`
: Special files for terminal interfaces

`/dev/xt/*`
: Special files for multiplex mode

`/etc/passwd`
: Information on login names

`/etc/ps_data`
: Contains internal data structures

`/proc`
: Process file system

SEE ALSO

`kill`(1), `nice`(1), `priocntl`(1), `renice`(1), `sh`(1), `getty`(1M), `clone`(3C).

ps(1-ucb)
BSD Compatibility Package

NAME

ps - display the status of current processes

SYNOPSIS

/usr/ucb/ps [–acglnrSuUvwx] [–t*term*] [*num*]

DESCRIPTION

The ps command displays information about processes. Normally, only those processes that are running with your effective user ID and are attached to a controlling terminal [see termio(7)] are shown. Additional categories of processes can be added to the display using various options. In particular, the –a option allows you to include processes that are not owned by you (that do not have your user ID), and the –x option allows you to include processes without control terminals. When you specify both –a and –x, you get processes owned by anyone, with or without a control terminal. The –r option restricts the list of processes printed to running and runnable processes.

ps displays the process ID, under PID; the control terminal (if any), under TT; the CPU time used by the process so far, including both user and system time, under TIME; the state of the process, under S; and finally, an indication of the COMMAND that is running.

The state is given by a single letter from the following:

- O Process is running on a processor.
- S Sleeping. Process is waiting for an event to complete.
- R Runnable. Process is on run queue.
- I Idle. Process is being created.
- Z Zombie state. Process terminated and parent not waiting.
- T Traced. Process stopped by a signal because parent is tracing it.
- X SXBRK state. Process is waiting for more primary memory.

OPTIONS

The following options must all be combined to form the first argument:

- –a Include information about processes owned by others.
- –c Display the command name, as stored internally in the system for purposes of accounting, rather than the command arguments, which are kept in the process' address space. This is more reliable, if less informative, since the process is free to destroy the latter information.
- –g Display all processes. Without this option, ps only prints interesting processes. Processes are deemed to be uninteresting if they are process group leaders. This normally eliminates top-level command interpreters and processes waiting for users to log in on free terminals.
- –l Display a long listing, with fields F, PPID, CP, PRI, NI, SZ, RSS and WCHAN as described below.
- –n Produce numerical output for some fields. In a user listing, the USER field is replaced by a UID field.
- –r Restrict output to running and runnable processes.
- –S Display accumulated CPU time used by this process and all of its inherited children.
- –u Display user-oriented output. This includes fields USER, SZ, RSS and START as described below.
- –U Update a private database where ps keeps system information (**system administrator only).**
- –v Display a version of the output containing virtual memory. This includes fields SIZE and RSS, described below.

`-w` Use a wide output format (132 columns rather than 80); if repeated, that is, `-ww`, use arbitrarily wide output. This information is used to decide how much of long commands to print.

`-x` Include processes with no controlling terminal.

`-t`*term*
List only process data associated with the terminal, *term*. Terminal identifiers may be specified in one of two forms: the device's file name (for example, `tty04` or `term/14`) or, if the device's file name starts with `tty`, just the digit identifier (for example, `04`).

num A process number may be given, in which case the output is restricted to that process. This option must be supplied last.

DISPLAY FORMATS

Fields that are not common to all output formats:

`USER` Name of the owner of the process.

`NI` Process scheduling increment [see `getpriority`(3) and `nice`(3C-ucb)].

`SIZE`
`SZ`
The combined size of the data and stack segments (in kilobyte units).

`RSS` Real memory (resident set) size of the process (in kilobyte units).

`UID` Numerical user-ID of process owner.

`PPID` Numerical ID of parent of process.

`CP` Short-term CPU utilization factor (used in scheduling).

`PRI` The priority of the process (higher numbers mean lower priority).

`START`
The starting time of the process, given in hours, minutes, and seconds. A process begun more than 24 hours before the `ps` inquiry is executed is given in months and days.

`WCHAN`
The address of an event for which the process is sleeping, or in `SXBRK` state (if blank, the process is running).

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

- `00` Process has terminated. Process table now available.
- `01` A system process, always in primary memory.
- `02` Parent is tracing process.
- `04` Tracing parent's signal has stopped process. Parent is waiting [see `ptrace`(2)].
- `08` Process is currently in primary memory.
- `10` Process currently in primary memory, locked until an event is completed.

A process that has exited and has a parent, but has not yet been waited for by the parent is marked `<defunct>`; otherwise, `ps` tries to determine the command name and arguments given when the process was created by examining the user block.

NOTES

Things can change while `ps` is running; the picture it gives is only a close approximation to the current state. Some data printed for defunct processes is irrelevant.

If no *term* or *num* is specified, `ps` checks the standard input, the standard output, and the standard error in that order, looking for the controlling terminal and will attempt to report on processes associated with the controlling terminal. In this situation, if the standard input, the standard output, and the standard error are all redirected, `ps` will not find a controlling terminal, so there will be no report.

On a heavily loaded system, `ps` may report an `lseek`(2) error and exit. `ps` may seek to an invalid user area address, having obtained the address of process' user area, `ps` may not be able to seek to that address

before the process exits and the address becomes invalid.

FILES

`/dev`
`/dev/sxt/*`
`/dev/tty*`
`/dev/xt/*`
: terminal (`tty`) names searcher files

`/dev/kmem`
: kernel virtual memory

`/dev/swap`
: default swap device

`/dev/mem`
: memory

`/etc/passwd`
: UID information supplier

`/etc/ps_data`
: internal data structure

SEE ALSO

`kill`(1), `whodo`(1M), `lseek`(2), `getpriority`(3C), `nice`(3C-ucb).

User's Reference Manual

pwd(1)

NAME

pwd - print path name of working directory

SYNOPSIS

```
pwd
```

DESCRIPTION

The shell built-in pwd writes the absolute path name of the working (current) directory on the standard output.

If the error message

```
Cannot open ...
```

or

```
Read error in ...
```

is displayed, a file system error has occurred. Inform the system administrator.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

You want to define your current directory as prg directory:

```
$ pwd
/usr/art/cobol/prg
$ prg=`pwd`; export prg
$ echo $prg
/usr/art/cobol/prg
```

NOTES

pwd exists both as an external command (/usr/bin/pwd) and as a built-in shell command, both in the Korn shell ksh(1) as well as in the Bourne shell sh(1). The shell generates a new process to execute /usr/bin/pwd.

Some differences in behavior may occur when using pwd, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

cd(1), ksh(1), sh(1).

random(1) - sum(1-ucb)

User's Reference Manual

random(1)

NAME

random - generates a random number

SYNOPSIS

`random` [`-s`] [*scale*]

DESCRIPTION

`random` generates a random number, *scale*, on standard output, and returns the number as its exit value. By default, this number is either 0 or 1. If *scale* is given a value between 1 and 255, the range of the random value is from 0 to *scale*. If *scale* is greater than 255, an error message is printed.

When the `-s` (silent) option is given, then the random number is returned as an exit value, but is not printed on the standard output. If an error occurs, `random` returns an exit value of zero.

NOTES

This command does not perform any floating point computations. `random` uses the time of day as a seed.

SEE ALSO

`rand`(3C) .

User's Reference Manual

read(1)

NAME

`read -` read arguments from standard input and assign to shellvariables

SYNOPSIS

`read` *name ...*

DESCRIPTION

`read` is a shell built-in command that reads a line from standard input and sequentially assigns the individual input line arguments as values to the shell variables specified in the call.

The only argument separators that `read` recognizes are the characters assigned to the `IFS` shell variable, the defaults being blanks, tabs and newline characters.

If `read` appears in a shell script and standard input has not been redirected, the script halts execution in order to read your next input from standard input. The script resumes execution as soon as you enter a newline character (see **EXAMPLES**).

OPERANDS

The Korn shell `ksh` supports additional options [see `ksh(1)`].

name Name of the shell variable to which the corresponding input line argument is assigned. The first argument is assigned to the first name, the second argument goes to the second, and so on, with the last name assigned whatever remains on the input line.

The names of shell variables must start with a letter or an underscore (_) and must consist of letters, underscores and digits only.

Any leftover arguments in the input line are assigned to the last variable specified in the `read` command line.

Excess variables are assigned the null string.

EXIT STATUS

`0` when `read` executes successfully

`>0` when no input is received, i.e. EOF is encountered.

ERROR MESSAGES

`text: not an identifier`

This error message may have the following causes:

- either you did not specify a variable name on the command line, or
- the name you specified contains illegal characters.

`read: missing arguments`

You called `read` without arguments.

ENVIRONMENT VARIABLES

`IFS` Input field separator (argument delimiter). The default values are blank, tab and newline.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The `read` command is invoked in a script named `readtest`, which contains the following:

```
: Invoked with sh readtest, halts for input
echo Please enter a customer name:
read customer1 customer2 customer3
if [ –z "$customer1" ]
then exit 5
else echo Customer1: $customer1
     echo Customer2: $customer2
     echo Customer3: $customer3
fi
```

Invocation of the `readtest` script file:

```
$ sh readtest
Please enter a customer name:
Shaw Bowden Pitman Potter
Customer1: Shaw
Customer2: Bowden
Customer3: Pitman Potter
```

After invocation, the shell script issues the message specified in the `echo` command and invokes `read`. The script halts, and the entered customer names are then read in.

The newline character terminates the input line for `read`. The third variable `customer3` is assigned two names, since four arguments were specified in the input line.

Example 2

Use of the `read` command to read in the first line from a file:

```
$ read line < /etc/group
$ echo $line
root::0:root
```

In this case, the first line of the file `/etc/group` will always be read, even if `read` is invoked repeatedly.

Example 3

The following shell script makes use of the `read` command in order to read in lines from a file successively:

```
: Invoked with sh readinall
exec < /etc/group
for i in 1 2 3 4 5 6 7
do
  read record$i
  eval echo record$i: \$record$i
done
```

In the shell script `readinall`, the shell built-in `exec` redirects the standard input to the file `/etc/group` for the following `read` command.

Owing to the `for` loop, `read` is invoked seven times in the script. Each invocation positions the read pointer on the next line, thus causing `echo` to output the first seven lines of the `/etc/group` file in succession:

```
$ sh readinall
record1: root::0:root
record2: daemon::1:daemon
record3: sys::2:sys:
record4: bin::3:bin,admin
record5: uucp::4:
record6: ces::5:
record7: other::10:gast,mgast,tele
```

To evaluate the argument `\$record$i` correctly, the shell has to interpret the `echo` command line twice;

hence the inclusion of `eval`. At the first attempt the shell only interprets `$i`, as the first dollar sign is escaped by the backslash. At the second attempt the shell interprets `$record[1-7]`.

NOTES

Some differences in behavior may occur when using `read`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`exec`(1), `ksh`(1), `sh`(1).

readonly(1)

NAME

`readonly -` protect shell variables

SYNOPSIS

`readonly` [*name* ...]

DESCRIPTION

The shell built-in `readonly` marks the specified shell variables as read-only, i.e. protects them from being changed by reassignment in the current shell. An error message is issued if any such attempt is subsequently made.

This protection only applies to the current shell. This means that the variable may be reassigned in a subshell or in a parent shell (i.e. when you terminate the current shell). Within the current shell, however, it is not possible to undo this protection.

If you invoke `readonly` without arguments, a list of all existing read-only variables in the current shell is written to standard output.

OPERANDS

The Korn shell `ksh` supports additional options, and a value can be assigned to the `readonly` variables [see `ksh`(1)].

name Name of the shell variable to be marked as read-only (i.e. protected). You have to assign a value to this variable before you call `readonly`.

You can list any number of shell variables, separated by commas.

name not specified:

All shell variables protected in the current shell are written to standard output in the following format:

`readonly` *name*

or for the Korn shell `ksh`:

name `=` *value*

or, if the `-p` option is set:

`readonly` *name* `=` *value*

ERROR MESSAGE

name`: is read only`

This error message is issued when you try to assign a value to a protected shell variable.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Marking the shell variable `HOME` as read-only in the current shell:

```
$ readonly HOME
$ readonly
readonly HOME
```

```
$ sh
$ readonly
$ CTRL-D
$ readonlyreadonly HOME
```

The `HOME` variable is not protected in the subshell.

NOTES

Some differences in behavior may occur when using `readonly`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh`(1), `sh`(1).

User's Reference Manual

red(1)

NAME

`red` - restricted interactive line editor

SYNOPSIS

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

DESCRIPTION

`red` is the same as `ed` with the following two restrictions: it can only edit files in the current working directory; and it does not allow the `!`*command* mechanism for the execution of shell commands.

See `ed`(1) for a description of the parameters.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In regular expressions in square brackets, the `LC_COLLATE` environment variable governs the scope of character ranges, equivalence classes and collating elements, and the `LC_CTYPE` environment variable governs the scope of character classes. `LC_CTYPE` also governs which characters the `ed` command `l` treats as non-printing.

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

The following session with `red` demonstrates and explains all the commands available in `red`. A fuller example is given under `ed`(1) .

red command	Explanation
`red`	Call red
`P`	Display `*` prompt in command mode
`*a`	Append to current line, in this case to beginning of file
`First line`	Input text
`Last line`	Input text
`.`	Terminate input mode
`*2 i`	Insert before second line
`Second line`	Input text
`.`	Terminate input mode
`*1,3 p`	Print lines 1 to 3
`First line` `Second line` `Last line`	
`*w textfile`	Write buffer contents to file named `textfile`
`33`	The number of characters written is displayed

`*q`	Quit `red`
`$`	Bourne shell prompt

SEE ALSO

`ed`(1) .

refer(1)
BSD Compatibility Package

NAME

`refer` - expand and insert references from a bibliographic database

SYNOPSIS

`/usr/ucb/refer` [`-b`] [`-e`] [`-n`] [`-ar`] [`-c`*string*] [`-k`*x*] [`-l`*m*,*n*] [`-p` *filename*] [`-s`*keys*] *filename* ...

DESCRIPTION

`refer` is a preprocessor for `nroff(1M)` , or `troff(1M)` , that finds and formats references. The input files (standard input by default) are copied to the standard output, except for lines between " `.[`" and "`.]`" command lines. Such lines are assumed to contain keywords as for `lookbib(1)` , and are replaced by information from a bibliographic data base. The user can avoid the search, override fields from it, or add new fields. The reference data, from whatever source, is assigned to a set of `troff` strings. Macro packages such as `ms(5)` print the finished reference text from these strings. A flag is placed in the text at the point of reference. By default, the references are indicated by numbers.

When `refer` is used with `eqn(1M)` , `neqn`, or `tbl(1M)` , `refer` should be used first in the sequence, to minimize the volume of data passed through pipes.

OPTIONS

`-b` Bare mode — do not put any flags in text (neither numbers or labels).

`-e` Accumulate references instead of leaving the references where encountered, until a sequence of the form:

```
.[
$LIST$
.]
```

is encountered, and then write out all references collected so far. Collapse references to the same source.

`-n` Do not search the default file.

`-a`*r* Reverse the first *r* author names (Jones, J. A. instead of J. A. Jones). If *r* is omitted, all author names are reversed.

`-c`*string*
Capitalize (with SMALL CAPS) the fields whose key-letters are in *string*.

`-k`*x* Instead of numbering references, use key labels as specified in a reference data line beginning with the characters `%`*x*; by default, `%`*x* is `%L`.

`-l`*m*,*n*
Instead of numbering references, use labels from the senior author's last name and the year of publication. Only the first *m* letters of the last name and the last *n* digits of the date are used. If either of *m* or *n* is omitted, the entire name or date, respectively, is used.

`-p` *filename*
Take the next argument as a file of references to be searched. The default file is searched last.

`-s`*keys*
Sort references by fields whose key-letters are in the *keys* string, and permute reference numbers in the text accordingly. Using this option implies the `-e` option. The key-letters in *keys* may be followed by a number indicating how many such fields are used, with a `+` sign taken as a very large number. The default is `AD`, which sorts on the senior author and date. To sort on all authors and then the date, for instance, use the options `-sA+T`.

FILES

`/usr/ucblib/reftools/papers`
: Default publication lists and indexes

`/usr/ucblib/reftools`
: Programs

SEE ALSO

`addbib`(1), `eqn`(1M), `indxbib`(1), `lookbib`(1), `nroff`(1M), `roffbib`(1), `sortbib`(1), `tbl`(1M), `troff`(1M).

User's Reference Manual

renice(1)

NAME

renice - alter the scheduling priority of running processes

SYNOPSIS

/usr/ucb/renice [–n *increment*] [–g | –p | –u] *ID* ...

/usr/ucb/renice *priority* [–p *pid* ...] [–g *gid* ...] [–u *user* ...]

DESCRIPTION

By default, each process has a priority of 20. The renice command alters the scheduling priority of one or more running processes. The relevant processes are usually specified using their process IDs. If a process group is specified, (see option –g), the request applies to all the processes in the process group.

If the renice command is executed with the –u option, the request is valid for all processes whose effective user ID corresponds with that of the specified user.

To determine if the user has the appropriate privileges, the real user ID under which renice is running is checked to see if it corresponds with the real or the effective user ID of the specified process.

Altering Process Priority

Users other than the privileged user may only alter the priority of processes they own, and can only monotonically increase their "nice value" within the range 0 to 20. This prevents overriding administrative fiats. The privileged user may alter the priority of any process and set the priority to any value in the range –20 to 20. Useful priorities are: 19 (the affected processes will run only when nothing else in the system wants to), 0 (the "base" scheduling priority) and any negative value (to make things go very fast). 20 is an acceptable nice value, but will be rounded down to 19.

OPTIONS

–n *increment*

Establishes how the scheduling priority of the specified process(es) should be adjusted. The *increment* argument is a positive or negative decimal integer. The scheduling priority of the specified process(es) is altered according to the *increment*.

Positive *increment* values lower the scheduling priority. Negative *increment* values require the appropriate privileges and increase the scheduling priority.

If the requested *increment* value raises or lowers the priority of the executed process over the limit value, the limit whose value was exceeded is used.

priority

Value range as for the *increment* parameter of the –n option, though negative values are reserved for use by the system administrator (root).

–g renice interprets all operands as process group IDs.

–p renice interprets all operands as process IDs. If no options are specified, the –p option is used by default.

–u renice interprets all operands as users. If a user exists whose user name is the same as the operand, the user ID of the user is used for further processing. Otherwise, if the operand represents an unsigned integer, this is used as the numeric user ID of the user.

ID A process ID, process group ID, or a user name/user ID, depending on the option selected.

ENVIRONMENT VARIABLES

The following environment variables affect execution of the renice command:

LANG Specifies a default value for the locale variables that are unset or null. If LANG is unset or null, the

corresponding default value from the locale is used. If the locale contains an invalid setting, the command behaves as if there were no variables defined.

`LC_ALL`
: If this variable is set, i.e. is not empty, this value overwrites the values of all other internationalization variables.

`LC_CTYPE`
: Governs the interpretation of byte sequences as characters (e.g. single-byte as opposed to multibyte characters in arguments).

`LC_MESSAGES`
: Governs the format and content of status messages.

`NLSPATH`
: Determines the location of message catalogs for the processing of `LC_MESSAGES`.

EXIT STATUS

`0` Successful completion.

`>0` An error occurred.

EXAMPLES

1. The process IDs 987 and 32 are assigned a lower priority:

   ```
   renice –n 5 –p 987 32
   ```

2. If the user has the appropriate privileges, the scheduling priority can be adjusted to assign the group IDs 324 and 76 a higher priority:

   ```
   renice –n –4 –g 324 76
   ```

3. The numeric user ID 8 and the user `sas` are assigned a lower priority:

   ```
   renice –n 4 –u 8 sas
   ```

SEE ALSO

`nice(1)`, `priocntl(1)`, `ps(1)`.

User's Reference Manual

rksh(1)

NAME

`rksh` - restricted Korn shell

SYNOPSIS

`rksh` [*option ...*] [*file*] [*argument ...*]

DESCRIPTION

`rksh` is a restricted version of the Korn shell. It is used to set up special login names and execution environments whose capabilities are more closely controlled than those of the standard environment.

OPTIONS

option
See `ksh`(1) .

file See `ksh`(1) .

argument
See `ksh`(1) .

Functionality

The functionality of `rksh` is identical to that of the Korn shell `ksh`, but with the following restrictions:

- The Korn shell built-in `cd` is not available, so you cannot change your current directory.
- You cannot change the values of the `ENV`, `PATH` and `SHELL` variables.
- Commands called with a command name containing a slash are not executed. In other words, you can only run commands which are located in your current directory or in one of the directories whose paths have been assigned to the `PATH` variable.
- Output cannot be redirected to files. If `>`, `>|`, `<>` or `>>` appears on the command line, the command will not be executed.

The above restrictions are enforced after `$HOME/.profile` and the files defined by the `ENV` variable have been interpreted.

When a command to be executed is found to be a shell script, `rksh` invokes the standard Korn shell `ksh` to execute it. Users are thus provided with access to the complete functionality of the Korn shell in scripts while working in a restricted environment.

For the system administrator only

As the system administrator, you can set up an environment for users who are only to be allowed to work in the restricted Korn shell. For such users you should define:

- the directory they are to work in.

  Users should not be allowed to work in the login directory and should only have read permission for it.
- the commands they should be allowed to use.

  You can create either a subdirectory in the login directory or a more general directory such as `/usr/rbin`, and copy to it the approved commands which can safely be invoked with `rksh`. The user should not have write permission for the directory of commands.
- the Korn shell scripts they should be allowed to use.

  It should not be possible to call a subshell, for example, since the subshell will not be restricted.
- the values of the `ENV`, `PATH`, and `SHELL` variables.

  You should, for example, ensure that an unrestricted shell cannot be called via the `!` command of the `ex` or `vi` editors.

You should enter the appropriate commands and assignments in the `$HOME/.profile` and `$ENV` files. The `cd` command that switches to the working directory in which the restricted Korn shell is to be used must be one of the last commands. Users should not have write access to these files or to the directories in which they are located (e.g. `$HOME`).

SEE ALSO

`ksh`(1) .

User's Reference Manual

rm(1)

NAME

`rm` - remove files

SYNOPSIS

`rm` [*option ...*] *file ...*

DESCRIPTION

`rm` removes the entry (link) for one or more files from a directory. You must have write permission for a directory before you can remove a file from it.

OPTIONS

No option specified:

If you have write permission for *file*, `rm` removes the entry without issuing a warning.

If you do not have write permission for *file* and the standard input is a terminal, `rm` prompts you by displaying the permissions and a query ? and asking whether you want the file to be deleted. The entry is not deleted unless you answer the locale's equivalent of yes. No such confirmation is requested if the standard input is not a terminal.

`-f` Removes the entries without any questions. Files will not be removed if you do not have write permission for the directory.

`-r` or `-R`

Accepts a directory name as an argument for *file*. The usual error message is not issued. `rm` recursively deletes the entire contents of the directory and also removes the directory itself.

You cannot remove the parent directory (`..`) in this way. Symbolic links that are encountered with this option are not traversed.

If the removal of a non-empty, write-protected directory is attempted, the command will always fail (even if the `-f` option is used), resulting in an error message.

`-i` `rm` removes files interactively, requesting confirmation for each write-protected file (or directory, if `-r` or `-R` is in effect) before removing it.

The `-i` option overrides the `-f` option and remains in effect even if the standard input is not a terminal.

`--` Two dashes explicitly mark the end of command-line options when there are file name arguments which begin with a dash.

file Name of the file that is to be removed. If you include the `-r` or `-R` option, *file* may also be a directory. You can include a number of file/directory arguments.

If you specify a file that has several links, only the specified link is removed; the file itself remains intact (the link counter is decremented by one). The file itself is not deleted until you remove the final link.

You must have write permission for the directory before you can remove a file from it; but you need not have read or write permission for the file itself.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

Answers to yes/no queries must be given in the language appropriate to the current locale.

EXAMPLES

Example 1

Removal of all files that end in `prog` with confirmation:

```
$ rm –i *.prog
exec.prog:? (y/n) y
code.prog:? (y/n) yuppie
input.prog:? (y/n) n
lucky.prog:? (y/n) no
a.prog:? (y/n) tomorrow
$
```

The links to files `exec.prog` and `code.prog` are removed; the others are retained.

Example 2

Removal of the directory `norm` with all files and subdirectories:

```
$ rm –r norm
```

SEE ALSO

`rmdir`(1), `unlink`(2), `remove`(3C).

User's Reference Manual

rmail(1)

NAME

`rmail` - send mail

SYNOPSIS

`rmail` [`-tw`] [`-m` *message_type*] *recipient* ...

DESCRIPTION

`rmail` is a restricted version of `mail(1)` that only permits the sending of mail. `uucp` uses `rmail` as a security precaution. If you are writing an application which generates mail messages, you should always use `rmail` instead of `mail`.

The functionality of `rmail` is identical to the functionality of `mail(1)` in send mode.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_CTYPE` governs character classes and character conversion (shifting).

`LC_COLLATE` governs the collating sequence.

The `LC_TIME` environment variable governs the format of date and time strings.

If `LC_MESSAGES`, `LC_CTYPE`, `LC_COLLATE` or `LC_TIME` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

SEE ALSO

`mail(1)`.

User's Reference Manual

rmail(1-ucb)

NAME

`rmail` - handle remote mail received via uucp

SYNOPSIS

`rmail` *user* ...

DESCRIPTION

`rmail` interprets incoming mail received via `uucp(1)`, collapsing "From" lines in the form generated by `mail.local(1M)` into a single line of the form "return-path!sender", and passing the processed mail on to `sendmail(1M)`.

`rmail` is explicitly designed for use with `uucp(1)` and `sendmail(1M)`.

SEE ALSO

`uucp(1)`, `mail.local(1M)`, `sendmail(1M)`.

User's Reference Manual

rmdir(1)

NAME

`rmdir -` remove directories

SYNOPSIS

`rmdir [ –p ] [ –s ] [ –– ]` *directory ...*

DESCRIPTION

`rmdir` removes one or more empty directories. Directories containing files cannot be removed with `rmdir`. To remove a directory together with everything it contains you can use the `rm` command, specifying option `–r` or `–R`.

OPTIONS

No option specified:
`rmdir` removes the specified directories.

`–p` (p – parents) The specified directory is removed, and all empty parent directories in the specified path are removed recursively. A message is then written to standard output if a part of the path still remains.

`–s` Suppresses the message written by the `–p` option.

`––` End of the list of options. Must be specified if *directory* begins with `–`.

directory
Name of the directory you want to remove. You can name any number of directories.

ERROR MESSAGES

`rmdir: dir1: Directory not empty`

You have attempted to use `rmdir` to remove a directory `dir1` which still has files in it.

You can use the `rm` command with option `–r` or `–R` to remove directories that contain files.

`rmdir: dir1: Directory does not exist`

The directory named `dir1` does not exist.

`rmdir: ../dir1: Can't remove current directory or ..`

The current directory or its parent cannot be removed. Change to the parent directory.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Removing the directories `pro` and `proc`.

The contents of your current working directory are:

```
drwxr-xr-x  11 susan    other       5720   Nov 18 14:16 ./
drwxr-xr-x  13 root     root        3380   Nov 04 11:48 ../
-rw-------   1 susan    other         79   Jul 19 14:21 .profile
-rwx------   1 susan    other        125   May 25 10:29 begin
drwx------   2 susan    other         32   Oct 11 15:36 pro/
drwx--x--x   2 susan    other         32   Nov 07 10:43 proc/
```

The directories `pro` and `proc` are empty, so you can remove them with `rmdir`.

```
$ rmdir pro proc
$ ls -lpa
drwxr-xr-x  11 susan    other       5720   Nov 18 14:16 ./
drwxr-xr-x  13 root     root        3380   Nov 04 11:48 ../
-rw-------   1 susan    other         79   Jul 19 14:21 .profile
-rwx------   1 susan    other        125   May 25 10:29 begin
```

Your current directory contains a directory `dvz1`. In `dvz1` there is another directory `dvz2`, and within that `dvz3`. `dvz1` also contains the file `file1`.

```
$ rmdir -p dvz1/dvz2/dvz3
rmdir: dvz1/dvz2/dvz3: dvz1 not removed; Directory not empty
```

SEE ALSO

rm(1), rmdir(2), unlink(2), remove(3C).

User's Reference Manual

roffbib(1)
BSD Compatibility Package

NAME

`roffbib` - format and print a bibliographic database

SYNOPSIS

`/usr/ucb/roffbib` [`-e`] [`-h`] [`-Q`] [`-x`] [`-m` *filename*] [`-n`*p*] [`-o`*list*] [`-r`*aN*] [`-s`*N*] [`-T`*term*] [*filename*] ...

DESCRIPTION

The `roffbib` command prints out all records in a bibliographic database, in bibliography format rather than as footnotes or endnotes. Generally it is used in conjunction with `sortbib(1)` :

```
example% sortbib database | roffbib
```

If abstracts or comments are entered following the `%X` field key, `roffbib` will format them into paragraphs for an annotated bibliography. Several `%X` fields may be given if several annotation paragraphs are desired.

`roffbib` accepts all options understood by `nroff(1M)` except `-i` and `-q`, as well as those listed below.

OPTIONS

`-e` Produce equally-spaced words in adjusted lines using full terminal resolution.

`-h` Use output tabs during horizontal spacing to speed output and reduce output character count. TAB settings are assumed to be every 8 nominal character widths.

`-Q` Queue output for the phototypesetter. Page offset is set to 1 inch.

`-x` Suppress printing of abstracts.

`-m` *filename*
Prepend the macro file `/usr/ucblib/doctools/tmac/tmac.name` to the input files. There should be a space between the `-m` and the macro filename. This set of macros will replace the ones defined in `/usr/ucblib/doctools/tmac/tmac.bib`.

`-n`*p* Number first generated page *p*.

`-o`*list*
Print only page numbers that appear in the comma-separated *list* of numbers and ranges. A range *N*–*M* means pages *N* through *M*; an initial –*N* means from the beginning to page *N*; a final *N*– means from page *N* to end.

`-r`*aN* Set register *a* (one-character) to *N*. The command-line argument `-rN1` will number the references starting at 1.

Four command-line registers control formatting style of the bibliography, much like the number registers of `ms(5)` . The flag `-rV2` will double space the bibliography, while `-rV1` will double space references but single space annotation paragraphs. The line length can be changed from the default 6.5 inches to 6 inches with the `-rL6i` argument, and the page offset can be set from the default of 0 to one inch by specifying `-rO1i` (capital O, not zero).

`-s`*N* Halt prior to every *N* pages for paper loading or changing (default *N* = 1). To resume, enter NEWLINE or RETURN.

`-T`*term*
Specify *term* as the terminal type.

NOTE

Users have to rewrite macros to create customized formats.

FILES

`/usr/ucblib/doctools/tmac/tmac.bib`
file of macros used by `nroff/troff`

SEE ALSO

`addbib`(1), `indxbib`(1), `lookbib`(1), `refer`(1), `sortbib`(1), `nroff`(1M), `troff`(1M).

User's Reference Manual

sar(1)

NAME

`sar` - system activity reporter

SYNOPSIS

Format 1

`sar` [*option ...*] [`−o` *file*] [`−P` [*no*] [*P_option ...*]] *t* [*n*]

Format 2

`sar` [*option ...*] [`−s` *time*] [`−e` *time*] [`−i` *sec*] [`−f` *file*] [`−C`] [`−P` [*no*] [*P_option ...*]]

DESCRIPTION

`sar` samples cumulative activity counters in the operating system at intervals which you specify. However, the `uname` information for the logged system is output before this data. If you name a file on the command line, the samples are saved in binary format in the specified file (**Format 1**). `sar` can also be used to selectively extract the data that was recorded in a file during a specified period (**Format 2**).

Format 1: Reporting system activity

`sar` [*option ...*] [`−o` *file*] [`−P` [*no*] [*P_option ...*]] *t* [*n*]

OPTIONS

No option specified:

`sar` reports on CPU utilization. The same information is output as under the `−u` option.

The following options can be used to specify the subsets of data to be printed:

`−a` `sar` reports on the use of file access system routines.

Header	Meaning
`iget/s`	Number of files located by inode entry per second.
`namei/s`	Number of file system path searches per second.
`dirbk/s`	At the moment this corresponds to `readdir/s` in `sar −x`. In the future: Number of UFS directory block reads issued per second.

`−A` `sar` reports all data (equivalent to specifying all the other options).

`−b` `sar` reports on buffer activity.

Header	Meaning
`bread/s`, `bwrit/s`	Transfers of data per second between system buffers and disk or other block devices.
`lread/s`, `lwrit/s`	Accesses to system buffers per second.
`%rcache`, `%wcache`	Cache hit ratios, where: `%rcache 1-bread/lread` (as a percentage), `%wcache 1-bwrit/lwrit` (as a percentage).
`pread/s`, `pwrit/s`	Number of physical transfers per second.

`−c` `sar` reports on system calls.

Header	Meaning
`scall/s`	All types of system call per second.

sread/s, swrit/s, fork/s, exec/s	Specific system calls per second.
rchar/s, wchar/s	Characters transferred by read() and write() system calls per second.

-d sar reports on the activity of the I/O devices, for which the system produces I/O statistics. These devices currently include:

- Physical hard disks (sdisk)
- RAID Luns (sraid)
- Virtual disks (vdisk)
- Magnetic tape drives (stape)

On a system that uses look-ahead caching for its disks, the value of %busy may be over 100%, especially when sequential data is being read.

The microsecond timer is initialized within five minutes of the system booting. The effect of this is that device statistics generated on the basis of the microsecond timer are either imprecise or even incorrect within this time period (only relevant for RM600).

Header	Meaning
%busy	Percentage of time device spent servicing a transfer request.
avque	Average number of requests outstanding during the monitored period.
r+w/s	Number of data transfers from or to device per second.
blks/s	Number of 512-byte blocks transferred per second.
avwait	Average time in milliseconds that transfer requests wait idly on queue.
avserv	Average time in milliseconds for transfer requests to be serviced (which for disks includes seek, rotational latency and data transfer times).

-g sar reports on paging activities.

Header	Meaning
pgout/s	Page-out requests per second.
ppgout/s	Pages paged-out per second.
pgfree/s	Pages per second placed on the free list by the page stealing daemon.
pgscan/s	Pages per second scanned by the page stealing daemon.
%ufsipf	Percentage of UFS inodes taken off the free list by iget which had reusable pages associated with them. These pages are flushed and cannot be reclaimed by processes.

-k sar reports on kernel memory allocation (KMA) activities.

Header	Meaning
sml_mem, alloc, fail	Information about the memory pool reserving and allocating space for small requests (< 256 bytes): the amount of memory in bytes KMA has for the small pool, the number of bytes allocated to satisfy requests for small amounts of memory, and the number of requests for small amounts of memory that were not satisfied.
lg_mem, alloc, fail	Information about the large memory pool (512 bytes to 4 Kbytes;

	analogous to the information for the small memory pool, see above).
ovsz_alloc, fail	Amount of memory allocated for oversize fail requests (> 4 Kbytes) and the number of oversize requests which could not be satisfied. Since oversized memory is allocated dynamically, there is no pool for it.

-K sar reports fully on kernel memory allocation (KMA) activities.

Header	Meaning
sml_mem, alloc, fail, pool	Information about the memory pool reserving and allocating space for small requests (< 256 bytes): the amount of memory in bytes KMA has for the small pool, the number of bytes allocated to satisfy requests for small amounts of memory, the number of requests for small amounts of memory that were not satisfied, and the number of pools in the category.
lg_mem, alloc, fail, pool	Information about the memory pool for large amounts of memory (512 bytes to 4 Kbytes; analogous to sml_mem, see above).
fst_mem, alloc, fail, pool	Information about the fast memory pool for large amounts of memory (512 bytes to 4 Kbytes; analogous to sml_mem, see above).
ovsz_mem, alloc, fail	Information about the memory pool (512 bytes to 4 Kbytes) reserving and allocating space for oversize requests (> 4 Kbytes), and the number of oversize requests which could not be satisfied. Since oversized memory is allocated dynamically, there is no pool for it.

Header	Meaning
fst_mem, alloc, fail, pool	Information about the fast memory pool for large sml_mem, see above).
ovsz_mem, alloc, fail	Information about the memory pool (512 bytes t (> 4 Kbytes), and the number of oversize reques allocated dynamically, there is no pool for it.

-l sar reports on locks.

Header	Meaning
splck/s, kslck/s	Overall number of spin locks and kernel locks per second.
wsplck, wkslck	Waits due to spin locks and kernel locks.

-m sar reports on message and semaphore activities.

Header	Meaning
msg/s	Number of message operations per second.
sema/s	Number of semaphore operations per second.

-p sar reports on paging activities.

Header	Meaning

atch/s	Page faults per second satisfied by reclaiming a page currently in memory (attaches per second).
pgin/s	Page-in requests per second.
ppgin/s	Pages paged-in per second.
vflt/s	Address translation page faults per second (validity faults).
pflt/s	Page faults from protection errors per second (illegal access to page).
slock/s	Faults per second caused by software lock requests requiring physical I/O.

-q sar reports on the average queue length while the queue is occupied and on the percentage of time the queue is occupied.

Header	Meaning
runq-sz, %runocc	Run queue of processes which are in memory and executable.
swpq-sz, %swpocc	Swap queue of processes which are swapped out and executable. These are no longer reported.

-r sar reports on unused memory pages and disk blocks.

Header	Meaning
freemem	Average number of pages available to user processes.
freeswap	Disk blocks available for page swapping.

-u sar reports on CPU utilization. This option is set by default.

Header	Meaning
%usr	Percentage of time running in user mode.
%sys	Percentage of time running in system mode.
%wio	Percentage of time idle with process waiting for block I/O.
%idle	Percentage of time idle.
int/s	Number of interrupts per second.
intdef/s	Number of delayed interrupts per second. □ In the current Reliant UNIX version this value is always 0.

-v sar reports on the status of the process, inode and file tables.

Header	Meaning
proc-sz, inod-sz, □ file-sz, lock-sz	Number of entries for each table, evaluated once at sampling point.
ov	Overflows that occur between sampling points for each table.

-w sar reports on system swapping and switching activity.

Header	Meaning
swpin/s, swpot/s, □ pswin/s, pswot/s	Number of transfers and number of 512-byte units transferred for swap-ins and swap-outs (including initial loading of some

	programs).
pswch/s	Process switches per second.

−x sar reports on the number of operations in the UFS file system.

Header	Meaning
□ open/s, □ create/s, □ lookup/s, □ readdir/s, □ getpage/s, □ putpage/s, □ other/s	Number of open, create, lookup, readdir, getpage, putpage, and other operations made per second.

−y sar reports on terminal device activity.

Header	Meaning
rawch/s	Input characters per second.
canch/s	Input characters processed by canon (canonical queue) per second.
outch/s	Output characters per second.
rcvin/s	Receiver hardware interrupts per second.
xmtin/s	Transmitter hardware interrupts per second.
mdmin/s	Modem interrupts per second.

−o *file*

The output of sar is saved in *file* in binary format.

−P [*no*] [*P_option* ...]

−P Output for CPUs on multiprocessor machines.

no *no* stands for the CPU number. If *no* is specified, −P only reports on the CPU with that number.

no not specified:

sar reports the values of the specified *P_options* for all CPUs.

P_option

−P can be combined with options −b, −c, −l, −m, −p, −w and −y. Descriptions of these options are given above under **Format□1**.

−P not specified:

sar reports an average value among all CPUs for the specified *options*.

t Time (in seconds) for which the activity counters are sampled. *t* should be greater than 5, as the activity of sar itself may otherwise affect the sample.

If *t* is specified with more than one option, the output may be difficult to read.

n Number of intervals in which activity counters are sampled for *t* seconds each.

n not specified:

The default value for *n* is 1.

Format 2: Displaying system activities recorded in a file

sar [*option* ...] [−s *time*] [−e *time*] [−i *sec*] [−f *file*] [−C] [−P [*no*] [*P_option* ...]]

OPTIONS

No option specified:

`sar` reports on CPU utilization. The same information is output as under the `-u` option.

option

See **Format 1**.

`-s` *time*

Starting time for the report, specified in the form: *hh*[:*mm*[:*ss*]].

`-e` *time*

Ending time for the report, specified in the form: *hh*[:*mm*[:*ss*]].

`-i` *sec*

Records are selected and reported at intervals of *sec* seconds.

`-i` not specified:

All intervals found in the daily activity file are reported.

`-f` *file*

Name of the file from which `sar` extracts data.

`-f` not specified:

`sar` reads the standard system daily activity file `/usr/adm/sa/sa`*dd*, where *dd* stands for the current day.

`-C` (only in conjunction with `-f` *file*) If the file was written with an older `sar` or `sadc` command, it can only be read with this option.

`-P` [*no*] [*P_option* ...]

See **Format 1**.

EXAMPLES

Example 1

To see today's CPU activity up to the time `sar` is called:

```
$ sar
```

Example 2

To watch system activity evolve for 12 minutes (12 periods of 60 seconds) and record the data in a file named `cpu_act`:

```
$ sar -o cpu_act 60 12
```

Example 3

To review I/O devices activity from the period recorded in *Example 2*:

```
$ sar -d -f cpu_act
```

NOTES

1. In the case of systems that support "look-ahead-caching" for their disks, the value for `%busy` (`-d` option) may be greater than 100 if a lot of data is read sequentially.
2. The `-b` option is used to report on the file system-specific access to all system buffers in the system, irrespective of the type of buffer.

FILES

`/usr/adm/sa/sa`*dd*

Daily data file for system activities, where *dd* stands for the current day.

SEE ALSO

`sar`(1M) .

User's Reference Manual

scp(1)

NAME

`scp` - secure copy (remote file copy program)

SYNOPSIS

`scp` [`-prvBC`] [`-P` *port*] [`-c` *cipher*] [`-i` *identity_file*] [[*user@*]*host1* `:`]*file1* [[*user@*]*host2* `:`]*file2*

DESCRIPTION

`scp` copies files between hosts in a network. It uses `ssh(1)` for data transfer. It also uses the same authentication and provides the same security as `ssh`. Unlike `rcp`, `ssh` prompts for passwords or passphrases if they are needed for authentication.

Any file name may contain a host and user specification to indicate that the file is to be copied to/from that host. Copies between two remote hosts are permitted.

OPTIONS

`-p` Preserves modification times, access times and modes from the original file.

`-r` Recursively copies entire directories.

`-v` Verbose mode. Causes `scp` and `ssh` to print debugging messages about their progress. This is helpful for debugging connection, authentication and configuration problems.

`-B` Selects batch mode (prevents prompting for passwords or passphrases).

`-C` Compression. Passes the `-C` flag to `ssh` to enable compression.

`-P` *port*
Specifies the port to connect to on the remote host. Note that this option is written with a capital P, because `-p` is already reserved for preserving the times and modes of the file in `rcp`.

`-c` *cipher*
Selects the cipher [see `ssh(1)`] to use for encrypting the data transfer. This option is directly passed to `ssh`.

`-i` *identity_file*
Selects the file from which the identity (private key) for RSA (Rivest-Shamir-Adelman) authentication is read. This option is directly passed to `ssh`.

SEE ALSO

`rcp(1)` , `ssh(1)` , `ssh-add(1)` , `ssh-agent(1)` , `ssh-keygen(1)` , `sshd(8)` .

User's Reference Manual

script(1)

NAME

`script -` make a typescript of a terminal session

SYNOPSIS

`script [ –a ] [ `*filename*` ]`

DESCRIPTION

`script` makes a typescript (i.e. a log file) of everything that is displayed on your terminal during a session. `script` ends when the forked shell exits or when you press <CTRL-D>.

OPTIONS

`–a` The typescript is appended to the contents of *filename*.

`–a` not specified:

A new typescript file is created.

filename

Log file in which the typescript is recorded.

file not specified:

The typescript is recorded in the file `typescript`.

EXAMPLES

```
$ script logfile
Script started, file is logfile
$ cat >prog.c
main()
{
   printf("Good Morning!\n")
}<CTRL-D>$ cc prog.c
c1:
prog.c    4: [syntax]:   ';' expected
c1: errors: 1, warnings: 0
$ <CTRL-D> Script done, file is logfile
$
```

`script` creates a file named `logfile` in the current directory. The first time you press <CTRL-D>, you are simply terminating the input for `cat`; the second <CTRL-D> terminates the typescript. *logfile* will then contain the following:

```
Script started on Thu Sep 6 11:57:51 1990
$ cat >prog.c
main()
{
   printf("Good Morning!\n")
}
$ cc prog.c
c1:
prog.c    4: [syntax]:   ';' expected
c1: errors: 1, warnings: 0
$
script done on Thu Sep 6 11:58:43 1990
```

User's Reference Manual

sdiff(1)

NAME

`sdiff -` side-by-side difference program

SYNOPSIS

`sdiff` [*option ...*] *file1 file2*

DESCRIPTION

`sdiff` displays the differences between two files on standard output. The differences are those found by the `diff`(1) command. `sdiff` prints a side-by-side listing of the two files on the standard output, marking the lines that are different. Lines from *file1* are displayed on the left, lines from *file2* on the right. Between the two files there is a gutter containing characters that indicate identical and differing lines. Identical lines have a blank gutter; lines that only exist in *file1* are marked by a < in the gutter, lines only in *file2* by a >. Lines which differ are marked | if there is an identical number of identical lines or | lines before them or between them in the two files.

OPTIONS

`–l` Prints only the left side (*file1*) of any lines that are identical.

`–o` *output_file*

output_file is the name of a third file that is created as a user-controlled merging of *file1* and *file2*. Identical lines of *file1* and *file2* are copied to *output_file* by default. Sets of differences, as produced by `sdiff`, are printed on the screen, a set being a group of differences sharing a common gutter character. After printing each set of differences, `sdiff` prompts you with a % and waits for you to type one of the following commands and terminate it with <RETURN>. Several such commands can be entered in immediate succession.

- `e l` Calls the editor `ed` to work on the left column of the last set of differing lines displayed.
- `e r` Calls the editor `ed` to work on the right column of the last set of differing lines displayed.
- `e b` Calls the editor `ed` to work successively on the left and right columns of the last set of differing lines displayed.
- `e` Calls the `ed` editor with an empty buffer.
- `l` Appends the left column of the output generated by `sdiff` to the contents of *output_file*.
- `q` Exits from the program.
- `r` Appends the right column of the output generated by `sdiff` to *output_file*.
- `s` Silent mode; does not print identical lines. Before printing the next set of differences, additionally creates an `ed`-script [see `ed`(1)] containing the `ed` commands that are needed to construct *file2* from *file1*.
- `v` Turns off option `s`.

On exit from the editor, the `ed` buffer is appended to the contents of *output_file*.

`–s` Does not print identical lines.

`–w` *n* Sets the number of columns for output lines to *n*.

`–w` *n* not specified:

The default width of an output line is 130 characters.

file1 file2

Names of the two files that `sdiff` is to compare. If you specify a directory for one of the two files, `sdiff` compares the second file with a file of the same name in the specified directory. `sdiff` issues an error message if it cannot find or access one of the files. You will need read permission for the files and execute permission for the directory [see `chmod`(1)].

If you use a dash – as the name for *file2*, `sdiff` reads from standard input and compares what it reads there with *file1*.

EXAMPLES

The files `file1` and `file2` have the following contents:

```
file1:     file2:
  x               y
  a               a
  b               d
  c               c
  d
```

The following output is produced by `sdiff`:

```
$ sdiff file1 file2
x       |       y
a               a
b       <
c       <
d               d
        >       c
```

SEE ALSO

`diff`(1), `ed`(1).

User's Reference Manual

sed(1)

NAME

`sed` - stream editor

SYNOPSIS

`sed` [`–n`] [[`–e`] *script* ...] [`–f` *scriptfile* ...] [`––`] [*file* ...]

DESCRIPTION

`sed` is a non-interactive stream editor that provides a similar set of functions to the interactive line editor `ed`.

`sed` is a versatile tool that allows you to:

- perform multiple global editing functions efficiently in one pass through the input,
- easily edit piped command output,
- apply a sequence of editing commands that is too complicated for interactive input.

`sed` reads files sequentially, edits each line with `sed` commands you have specified in a `sed` script, and writes the results on the standard output. The `sed` script is either read straight from the command line or taken from a file. `sed` acts as a filter, i.e. it does not change the original input file. If you want to save the changes, you will have to redirect standard output to a file.

OPTIONS

`–n` Suppresses the default output, which is to pass every input line to the standard output after processing (see *Example 6*).

`–n` not specified:

`sed` copies each processed input line to standard output, even if the `sed` script does not contain an output command such as `p`. The commands in the `sed` script determine whether a line is modified by editing instructions. Input lines that are processed by an output command such as `p` are thus shown twice in succession. First, as part of the default output of all processed lines, and then as the output of the special editing command which causes them to be displayed again.

[`–e`] *script*

Uses the command-line *script* to edit the input file. If the script contains blanks, newlines, or shell metacharacters, it must be enclosed in single quotes: '*script*'.

The `–e` *script* option may be included a number of times with different scripts and may also be combined with the `–f` *scriptfile* option. In this case `sed` applies the commands from all scripts on each input line.

If the command line contains the `–e` option only once and does not include option `–f`, you may omit the `–e` when you specify *script*. According to *script*, no more options or `––` may be specified.

`–f` *scriptfile*

Reads editing instructions for the input file from the file named *scriptfile*.

You can include the `–f` *scriptfile* option a number of times with different script files and also combine it with `–e` *script*. `sed` then reads the `sed` commands from all specified `sed` scripts.

`––` End of the list of options. Must be specified if *file* begins with – or if *script* begins with – and `–e` was not specified.

file Name of the file whose contents are to be processed by `sed`. The file must be a text file.

file not specified: `sed` reads from standard input.

Functionality

`sed` works on copies of the input lines, successively copying each line into a temporary work area called a "pattern space".

Each line of input is normally processed in the following cycle:

Step 1:
The next (or first) line of input is copied into the pattern space.

Step 2:
All `sed` script commands that select the pattern space (i.e. address the last line copied into it) are successively applied to its contents. Depending on the editing instruction in the script, the contents of the pattern space are then altered as required, or left unchanged.

Step 3:
The contents of the edited pattern space are then sent to standard output, and the pattern space is cleared.

This cycle is repeated until all input lines have been processed.

The pattern space may occasionally contain multiple lines (see commands `g`, `G`, `N`). However, it is always the address of the last line copied into the pattern space that serves as the pattern space address.

Some `sed` commands (`g`, `G`, `h`, `H`) copy text from the pattern space to a temporary storage area called the "hold space". The hold space saves all or part of the pattern space for subsequent retrieval.

Format of a sed script

A `sed` script consists of command lines in the form:

[*range*]*sed_command*[*flags* ...]

or

[*range*]{*sed_command*[*flags* ...]

sed_command[*flags* ...]

.□
.□
.

}

The braces are required only if you specify an address *range*.

The right brace must be positioned at the start of a line, i.e. only preceded by blanks or tabs.

No blanks are permitted between the *range* and the *sed_command*.

A *range* serves to select particular input lines and can be specified as one or two comma-separated addresses. When the given *range* "selects" the pattern space, i.e. addresses the last line copied to it, the associated `sed` command or command list is applied to the pattern space.

range = *address*
The pattern space that matches *address* is selected.

range = *address1*,*address2*
An inclusive range from the first pattern space that matches *address1* through the next pattern space that matches *address2* is selected. However, if *address2* is a line number in the input file that comes before the one selected with *address1*, only one line is selected (*address1*).

range not specified:

The pattern space is always selected, i.e. the last input line copied to it is always addressed.

Addresses

Address	Meaning
`$`	last line of the last input file
`n`	*n*th input line, where *n* is a positive integer. Input lines are numbered consecutively across all files.
/*pattern*/	Input line containing a string matching the specified *pattern*. Any slash / appearing

	in pattern must be preceded by a backslash \ to escape it. *pattern* is a simple regular expression [see `expressions(5)`] with the following modifications: □ The construction \?*regular_expression*?, where ? is any character, is identical to */regular_expression/*. Note that in the address that in the address, for example, the second x is escaped and stands for itself, so that the regular expression is abcxdef. □ The escape sequence `\n` matches a newline embedded in a pattern space. □ A period matches any character except the terminal newline of the pattern space.

sed commands

The following section contains a list of `sed` commands, described in alphabetical order. The square brackets [ ] are not to be entered; they merely indicate that the enclosed address or address range is optional.

The *text* argument consists of one or more lines. All but the final line must end with a backslash \ to escape the terminating newline.

[*address*]`a\`
text

Append *text* to the pattern space that is output.

[*range*]`b`[*label*]

Branch to the `sed` command `:`*label* in the `sed` script.

label not specified: Branch to the end of the `sed` script.

[*range*]`c\`
text

Change. Delete the selected pattern space, send *text* to the output and start the next cycle.

[*range*]`d`

Delete the contents of the pattern space and start the next cycle. Step 3 is dropped, i.e. the contents of the pattern space are not sent to standard output.

[*range*]`D`

Delete the initial segment of the pattern space up to (and including) the first newline and start the next cycle.

[*range*]`g`

Replace the contents of the pattern space by the contents of the hold space.

[*range*]`G`

Append the contents of the hold space to the pattern space.

[*range*]`h`

Replace the contents of the hold space by the contents of the pattern space.

[*range*]`H`

Append the contents of the pattern space to the hold space.

[*address*]`i\`
text

Insert *text* into standard output before the contents of the pattern space.

[*range*]`l`

List the pattern space on standard output, representing non-printing characters with replacement characters (e.g. tab characters; see `ed` command `l`) or with their two-digit octal ASCII code equivalents in the form `\`*nn*. Long lines with more than 71 characters are split into two or more lines (folded). A backslash at the end of a screen line indicates that the text line continues in the next output line. The end of each line is indicated by `$`.

[*range*]`n`

Next. Copy the contents of the pattern space to standard output and replace them with the next line of input. The address of the last line of input becomes the address of the pattern space.

[*range*]N
: Append the next line of input to the pattern space with an embedded newline. The address of the last appended line becomes the address of the pattern space.

[*range*]p
: Print the contents of the pattern space on standard output. Non-printing characters are not represented.

[*range*]P
: Print the initial segment of the pattern space, up to the first newline, on standard output. Non-printing characters are not represented.

[*address*]q
: Quit sed. If you have specified multiple sed scripts, sed quits at the first q encountered in any of the scripts.

[*range*]r *rfile*
: Read the contents of *rfile* and send them to standard output before copying the next input line to the pattern space. *rfile* must be separated from the sed command r by exactly one space and must come at the end of the command line.

[*range*]s/*RE*/*repstring*/[*flags*]
: Substitute *repstring* for strings that match the regular expression *RE* in the pattern space. *RE* can be specified in the form of a simple regular expression [see expressions(5)]. The delimiter does not have to be /: most other characters are accepted. For a fuller description see the s command in ed.

  flags

  n
  : Substitute *repstring* for just the *n*th instance of *RE* in a line.

  g
  : Globally substitute *repstring* for all instances of *RE* in a line.

  p
  : Print the contents of the pattern space on standard output if a replacement was made. This applies even if sed was invoked with the –n option.

  w *wfile*
  : Write. Write the pattern space to the file *wfile* whenever a replacement is made. Any file named *wfile* that was already present before you called sed will be overwritten. If a number of w commands in a sed script write to the same *wfile*, the contents of the pattern space will be appended to the existing contents of *wfile* in each case. *wfile* must be separated from the sed command w by exactly one space and must come at the end of the command line. A maximum of 10 different files can be used for *wfile* in any one invocation of sed.

    Caution:
    : If you use the name of your input file as *wfile*, you will destroy your input file!

  No *flags* specified:

  Only the first instance of *RE* in the line is replaced by *repstring*.

[*range*]t*label*
: Test. Branch to the sed command :*label* in the sed script if any substitutions have been made since the last input line was copied to the pattern space or since the most recent execution of a t command.

  label not specified: Branch to the end of the sed script.

[*range*]w *wfile*
: Write. Write the pattern space to the file *wfile*. Any file named *wfile* that was already present before you called sed will be overwritten. If a number of w commands in a sed script write to the same *wfile*, the contents of the pattern space will be appended to the existing contents of *wfile* in each case. *wfile* must be separated from the sed command w by exactly one space and must come at the end of the command line. A maximum of 10 different files can be used for *wfile* in any one invocation of sed.

  Caution:
  : If you use the name of your input file as *wfile*, you will destroy your input file!

[*range*]x
Exchange the contents of the pattern and hold spaces.

[*range*]y/*string1*/*string2*/
Transform. Replace each occurrence of a character in *string1* with the corresponding character in *string2*. *string1* and *string2* must be of the same length and must be specified explicitly. Regular expressions cannot be used.

[*range*]!*command*
Don't command. Apply *command* to all lines *not* selected by the specified address *range*. *command* may be any sed command or a sed command list enclosed in braces {...}.

:*label*
Set a *label* for b and t commands to branch to. *label* is any string up to 8 characters long.

[*address*]=
Prints the current line number on standard output on its own line.

[*range*]{*sed_command*
sed_command .□
.□
.□
}
Successively execute all sed commands enclosed within the braces if the address *range* selects the current pattern space. The right brace } must be positioned at the start of a line, i.e. only preceded by blanks or tabs.

<RETURN>
The newline character is treated as a null command and is ignored. This allows you to produce more transparent sed scripts by using blank lines.

If # is the first character entered in a line of a script file, the entire line is interpreted as a comment.

#n If the first line of a script file begins with the character sequence #n, the default output of the pattern space is suppressed (as with the –n option). The entire line is treated as a comment, i.e. not interpreted as an sed command.

ERROR MESSAGES

`sed: command garbled: ...`

The sed script contains a syntax error. The colon is followed by the script location at which sed terminated.

`Can't open file`

You have specified a nonexistent input file or a file for which you do not have read permission.

`Unrecognized command: ...`

The sed script contains an unknown command.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

In regular expressions in square brackets, the LC_COLLATE environment variable governs the scope of character ranges, equivalence classes and collating elements, and the LC_TYPE environment variable governs the scope of character classes. LC_TYPE also governs which characters the sed command l treats as non-printing.

If LC_MESSAGES, LC_COLLATE or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Write the string XXXXX in all blank lines of a file and redirect the output to another file:

```
$ sed '/^$/s/^/XXXXX/'file > newfile
```

/^$/ selects all blank lines, i.e. lines that contain nothing (not even a blank) between start of line and end of line. sed searches for the beginning of the line (^) and replaces it with the string XXXXX.

Example 2

Indent by 4 spaces all lines that begin with a digit, and redirect the output to another file:

```
$ sed '/^[0-9]/s/^/ERRORERRORERRORERROR/'file > newfile
```

/^[0-9]/ selects all lines that begin with a digit. sed locates the start of the line (^) and replaces it with 4 blanks ERRORERRORERRORERROR, i.e. shifts the rest of the line 4 positions to the right, padding positions 1 to 4 with spaces.

Example 3

Print all the non-blank lines in a file:

```
$ sed '/^$/d'file
```

All blank lines are selected by /^$/ and deleted by d.

Example 4

Make a file double-spaced by adding a blank line after each line in the file:

```
$ sed 's/$/\ <CR>> /' file
```

Since no address is specified, sed searches for the end of the line ($) in every line and replaces it with a newline character <CR>, thus adding a blank line after every line.

Example 5

Print the second and third column of a file in which individual columns are separated from one another by a colon. The third column is to be placed before the second:

```
$ sed 's/[^:]*:\([^:]*\):\([^:]*\):.*/\2:\1/'file
```

Explanation:

`s/ / /` The string between the first and second slash is searched for in each line and replaced by the string between the second and third slash.

`[^:]*` Any number (*) of characters other than a colon ([^:])

`:` Colon

`\( \)` Brackets off part of expression for reuse in the replacement string which is between the second and third slash.

`\2` The replacement string is to begin with the part of the expression which is in the second bracket \(...\).

: There is to be a colon between the first and second part of the replacement string.

`\1` The second part of the replacement string is to be the part of the expression which is in the first bracket \(...\).

Example 6

Filter all professions out of the file personnel and write them into the file named professions under a new heading. The following example works on the assumption that the used search key (Profession:) does not appear in the first line of the file.

The personnel file has the following format:

```
Name: John Miller
Marital status: Divorced
Profession : Journalist
```

```
Name: Catherine Baker
Marital status: Married
Profession: Programmer
etc.
```

sed program:

```
$ sed –n '1{s/^.*/Professions:/
            h
           }
          /^Profession:/{s/^Profession: *\(.*\)/\1/
                         H
           }
          ${g
            p
           }' personnel > professions
$ cat professions
Professions:
Journalist
Programmer
etc.
$
```

Explanation:

Line 1 in the pattern space is replaced by the string `Professions:` and then stored in the hold space by the `h` command.

Each line in the pattern space that begins with `Profession:` is replaced by the string that follows it and then appended to the hold space by the `H` command.

The last line of the file (`$`) is replaced in the pattern space by the entire contents of the hold space. Finally the pattern space is printed on standard output by the `p` command.

The `–n` option changes the default output so that the input lines copied to the pattern space are not automatically sent to standard output after editing. Only the `p` command actually prints the pattern space.

Example 7

The `protocol` file has the following contents:

```
First Line
RCS file:         RCS/awk,v;   Working file:    awk
head:             1.8
locks:            ;  strict
symbolic names:   Y4A0010: 1.8;  N4A0010: 1.8;  Y3C1001: 1.7;
comment leader:   "# "
total revisions: 8;    selected revisions: 8
----------------------------
Last Line
```

A `sed` script called `cutoff` can be used

1) to delete all lines between `First Line` and `Last Line`;

2) to insert a new `BEGIN` line before `First Line`;

3) to add a new `END` line after `Last Line`.

The `cutoff` script has the following contents:

```
#!/usr/bin/ksh
# Call: cutoff $1
       f=$1
       echo $f
{
```

```
         sed '/^RCS *\(.*\)/,/^----------------------------/d□
        /^First Line/i\□
BEGIN□
        /^Last Line/a\□
END□
      ' $f□
         } > $f.out□
mv $f.out $f
```

Explanation:

The

```
/^RCS *\(.*\)/,/^----------------------------/d
```

statement marks the area to be deleted. It starts with a line beginning with `RCS` followed by any sequence of characters [= `*\(.*\)`]. The area to be deleted ends with the line that begins with the character string `----------------------------`. These two lines as well as all lines in between are deleted. The

```
/^First Line/i\□
□
□
BEGIN
```

statement inserts a line with the character string `BEGIN` before a line beginning with `First Line`.

The

```
/^Last Line/a\□
□
□
END
```

statement adds a line with the character string `END` after a line beginning with `Last Line`.

Once the `cutoff` script has been executed, the `protocol` file has the following contents:

```
BEGIN□
First Line□
Last Line□
END
```

SEE ALSO

awk(1), ed(1), grep(1), expressions(5).

NAME

`set` - set shell options or positional parameters

SYNOPSIS

`set` [*option ...*] [*argument ...*]

DESCRIPTION

The shell built-in command `set` has three functions:

- If no operand is specified, `set` writes on standard output the names and values of all shell variables defined for the current shell.

  By contrast, the `env` command writes on standard output the names and values of all shell variables available to all commands and all subshells: variables such as `IFS`, `MAILCHECK`, `PATH`, `PS1` and `PS2` are not included unless they have been exported (with `export`).
- Any options assigned to `set` govern the behavior of the current shell.

  The same options are also available for the `sh` or `ksh` command, but in this case they govern the behavior of the subshell thus generated.
- Arguments used instead of an option or after the option list are handled by `set` as follows:
  - it assigns the first nine arguments to positional parameters `$1` through `$9`,
  - it assigns all the arguments to the shell parameters `$*` and `$@`,
  - it assigns the number of arguments to the shell parameter `$#`.

  If there are fewer than nine arguments, the remaining positional parameters are set to the null string.

  To access the tenth and subsequent command line arguments directly you need to use the `shift` command.

If you want to use `set` to control the execution of a shell script, you have to include `set` in the script. This is because shell scripts are always executed by a subshell.

OPTIONS

No option specified:

`set` sequentially assigns the first nine arguments to positional parameters `$1` through `$9`. It assigns all command line arguments collectively to the shell parameters `$@` and `$*`; and it assigns the total number of arguments to the shell parameter `$#`. If there are fewer than nine arguments, the remaining positional parameters are set to the null string.

To access the tenth and subsequent command line arguments directly you need to use the `shift` command.

No operand specified:

`set` writes on standard output the names and values of all shell variables defined for the current shell.

option

The options govern the behavior of the current shell.

If you want to use more than one option, you have to group the code letters together with no spaces between them and with a single minus sign at the beginning.

The `--` option cannot be combined with other options.

You can use the `echo $-` command to find out which options have already been set with the `set` or `sh` or `ksh` commands. Set options other than `-n` and `-t` can also be unset without the need to terminate the current shell.

The Korn shell `ksh` has additional options [see `ksh(1)`].

The following options are available:

`-a` Automatically exports all shell variables that you define or redefine in the current shell.

`-a` not specified and not set previously:

Shell variables that you define or redefine in the current shell are not automatically exported. In other words, a shell variable will not be known in a subshell unless you explicitly export it using the shell built-in `export`.

`+a` Turns off option `-a`. Any shell variables defined earlier while the `-a` option was active will continue to be exported automatically.

`-e` The current shell exits immediately if a command returns a non-zero exit status.

`-e` not specified and not set previously:

An interactive shell exits when you press CTRL-D. A shell running a shell script only exits prematurely if it encounters a syntax error. Otherwise it exits on reaching the end of the script, regardless of the exit status of individual commands.

`+e` Turns off option `-e`.

`-f` Disables the file name generation mechanism. The characters `*`, `?` and `[...]` are not treated as metacharacters and expanded to generate all the matching file names.

`-f` not specified and not set previously:

Matching file names are substituted for the above patterns.

`+f` Turns off option `-f`.

`-h` This option changes the behavior of the current shell when you define shell functions: The commands you include in a shell function are entered in the hash table at the time of function definition [see `hash(1)`].

`-h` not specified and not set previously:

Commands used in a shell function are not entered in the hash table until the function is executed.

`+h` Turns off option `-h`.

`-k` (Keyword) Variable assignments, i.e. arguments in the form *name=value*, are allowed at any point on the command line. The shell performs the assignment and makes the resultant keyword parameter available to the environment of the command (see *Example 1*).

`-k` not specified and not set previously:

Variable assignments must precede the command name. The shell makes the appropriate keyword parameters available to the environment of the command.

`+k` Turns off option `-k`.

`-n` The current shell reads and interprets all commands but does not execute them, thus omitting the last step in the processing of a command line [see `sh(1)` , **COMMAND LINE PROCESSING**].

The `-n` option can be used to check a shell script for syntax errors.

Caution:

If you enter `set -n` interactively, the only way to turn off the `-n` option is to exit the shell with CTRL-D.

The `-n` option cannot be turned off within a shell script. It is turned off automatically when the shell running the script exits.

`-n` not specified and not set previously:

All commands are read, interpreted, and executed.

`-t` Terminate. The current shell exits after executing one command, i.e. the command line that contains the `set -t` command.

`-t` not specified:

An interactive shell exits when you press CTRL-D. A shell running a shell script only exits prematurely if it encounters a syntax error. Otherwise it exits on reaching the end of the script.

If the `–e` option is set in the current shell, this shell exits immediately if a command returns a non-zero exit status.

`–u` Causes the current shell to issue an error message on encountering an unset shell variable in a command line. The command in question is not executed.

Shell scripts are terminated as soon as the shell encounters an undefined shell variable.

`–u` not specified and not set previously:

The shell does not issue an error message for an unset shell variable, but substitutes the null string as its value by default.

`+u` Turns off option `–u`.

`–v` Verbose. The current shell displays each subsequent input on standard error before executing the corresponding command.

In combination with the `–x` option, this option is useful for debugging shell scripts.

`–v` not specified and not set previously:

The current shell does not echo its input.

`+v` Turns off option `–v`.

`–x` The current shell interprets its input and writes it on standard error, marking each command that it processes with a plus sign (+) at the start of the line. A command consisting solely of a variable assignment is output without the plus sign. The shell then executes the command.

In contrast to the output with option `–v`, the `–x` option shows you how the shell has replaced metacharacters and shell variables.

Thus you can test a shell script as follows:

- either include `set –xv` as the first command in your shell script, or
- run the shell script with the command `sh –xv script`.

`–x` not specified and not set previously:

The current shell does not echo its input.

`+x` Turns off option `–x`.

`––` Cannot be combined with other options.

The first argument specified after `––` is not interpreted as an option, even if it begins with a minus sign. This enables you to use `set` to assign values beginning with a minus sign to positional parameters (see *Example 4*).

`set ––` has no effect on any other options currently in effect. In the Korn shell `ksh` however, all positional parameters are deleted.

argument

Any string delimited by blanks or tabs. The last argument is terminated by a command separator.

You can include any number of arguments, with at least one blank or tab between them.

`set` sequentially assigns the first nine arguments to positional parameters `$1` through `$9`. It assigns all command line arguments collectively to the shell parameters `$@` and `$*`; and it assigns the total number of arguments to the shell parameter `$#`. If there are fewer than nine arguments, the remaining positional parameters are set to the null string.

To access the tenth and subsequent command line arguments directly you need to use the `shift` command.

argument not specified:

The positional parameters `$1` through `$9` and the shell parameters `$@`, `$*` and `$#` are left unchanged.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables which affect internationalization.

EXAMPLES

Example 1

The shell script archive contains the following:

```
set | grep customer
echo $customer
```

The –k option is set in the current shell, and the shell script is then executed as follows:

```
$ set –k
$ sh archive customer=Meyer
customer=Meyer
Meyer
```

After set –k, the shell variable customer may be defined after the script name in the command line. This variable, which is accessed in the example by the echo and set commands, is only known in the subshell that runs the script.

When the script finishes, the subshell that runs it also exits. The variable customer is not known in the (now current) parent shell. The command below will therefore output nothing:

```
$ set | grep customer
```

If the –k option is now turned off with +k and the shell script is invoked as before, the customer variable will be undefined within the script. The echo command simply outputs a blank line.

In other words, if the –k option is not set, the variable definition must precede the script name on the command line.

```
$ set +k
$ sh archive customer=Meyer
$ customer=Meyer sh archive
customer=Meyer
Meyer
```

Example 2

The shell script evaltest has the following contents:

```
set –xv
eval echo \$$#
echo $0
```

The set command sets the –x and –v options in the subshell that executes this shell script. The shell script is then called as shown below:

```
$ sh evaltest today is friday
eval echo \$$#
+ eval echo $3
+ echo friday
friday
echo $0
+ echo evaltest
evaltest
```

The –v option (verbose) causes each command to be displayed before it is processed.

The –x option causes the shell to interpret each command line and output it with a + in the first column. The output shows that the command line arguments of the eval command are interpreted twice by the shell.

Example 3

The tape of a mounted tape cartridge is not to be retensioned before the first access. The mt(1) command takes the name of the special file from the shell variable TAPE.

The contents of the corresponding shell script mtno are given below:

```
set –u
mt –f $TAPE noret
```

Since the –u option is set, the mt command is executed only if the shell variable has been defined. This enables you to prevent mt from accessing the default special file /dev/rmt0.

If the TAPE variable has not been defined, the shell script will terminate immediately with the following error message:

```
$ sh mtno
mtno: TAPE: parameter not set
```

Example 4

The following shell script counts all arguments read from standard input. It is, however, somewhat slower than the wc –w command:

```
1  : sh count counts all arguments on standard input
2  num=0
3  while read line
4  do
5     set –– $line
6     num=`expr $num + $#`
7  done
8  echo $num
```

Line 3:

The shell built-in read reads a line from standard input and assigns it to the variable line.

Line 5:

The shell built-in set assigns the arguments which make up the value of the variable line to the positional parameters $1, $2 and so on; $# is the total number of arguments in $line.

In this example the –– option has two functions:

- Even if $line is empty, set still does not output the current environment. (In the Korn shell ksh, all available positional parameters are deleted.)
- If $line starts with a minus sign, set does not interpret its first argument as an option.

Line 6:

For each line that is read, expr increments the value of the num variable by the number of arguments in the input line ($#).

The commands in lines 5 and 6 are repeated until the last input line has been read in by the read command. The echo command then outputs the value of the num variable, which is the total number of arguments read (total number of words).

NOTES

set exists both as an external command (/usr/bin/sh) and as a built-in shell command, both in the Korn shell ksh(1) as well as in the Bourne shell sh(1). The shell generates a new process to execute /usr/bin/sh.

Some differences in behavior may occur when using set, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

env(1), getopt(1), ksh(1), sh(1).

User's Reference Manual

setext(1)
VxFS

NAME

`setext -` set extent attributes

SYNOPSIS

`setext [–e` *extent_size* `][–r` *reservation* `][–f align][–f contig][–f noextend][–f noreserve][–f trim][–f chgsize]` *file*

DESCRIPTION

`setext` allows space to be reserved to a file, and a fixed extent size to be specified for a file. The file must exist.

OPTIONS

`–e` *extent_size*
: Specify a fixed extent size. The extent size is specified in file system blocks.

`–r` *reservation*
: Preallocate space for a file. The reservation is specified in file system blocks.

`–f align`
: Specify that all extents must be aligned on *extent_size* boundaries relative to the start of allocation units.

`–f contig`
: Specify that the reservation must be allocated contiguously.

`–f noextend`
: Specify that the file may not be extended once the preallocated space has been used.

`–f noreserve`
: The reservation is to be made as a nonpersistent allocation to the file. The on-disk inode will not updated with the reservation information so that the reservation will not survive a system crash. The reservation is associated with the file until the close of the file. The reservation is trimmed to the current file size on close.

`–f trim`
: Specify that the file is to be trimmed to the current file size upon last close by all processes that have the file open.

`–f chgsize`
: Specify that the reservation is to be immediately incorporated into the file. The file's on-disk inode is updated to with size and block count information that is increased to include the reserved space.

  `chgsize` is restricted to the superuser.

SEE ALSO

`getext`(1), `vxfsio`(7).

User's Reference Manual

settime(1)

NAME

`settime -` changes the access and modification dates of files

SYNOPSIS

`settime` *mmddhhmm*[*yy*] [`-f` *fname*] *name* ...

DESCRIPTION

Sets the access and modification dates for one or more files. The dates are set to the specified date, or to the access and modification dates of the file specified via `-f`. Exactly one of these methods must be used to specify the new date(s). The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *yy* is the last two digits of the year and is optional. For example:

```
settime 1008004583 ralph pete
```

sets the access and modification dates of files `ralph` and `pete` to Oct. 8, 12:45 AM, 1983. Another example:

```
settime -f ralph john
```

This sets the access and modification dates of the file, `john`, to those of the file, `ralph`.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

NOTES

Use of `touch(1)` in place of `settime` is encouraged.

SEE ALSO

`touch(1)` .

sh(1)

NAME

`sh, bg, break, chdir, continue, fg, jobs, return, stop, suspend -` Bourne shell, acommand interpreter and programming language

SYNOPSIS

`sh` [*option ...*] [*file*] [*argument ...*]

DESCRIPTION

The description of the Bourne shell is divided into three main parts:

- the `sh` command
  - login shell and subshells
  - options
- the shell as command interpreter
  - entering and grouping commands
  - shell metanotation
  - shell variables
  - environment
  - shell parameters
  - command substitution
  - input/output redirection
  - file name generation
  - command line processing
- the shell as programming language
  - comments in shell scripts
  - shell functions
  - shell control-flow statements

THE SH COMMAND

The shell is the user interface to the operating system kernel. It is implemented as a Reliant UNIX command and is not part of the kernel; the shell and all other Reliant UNIX commands communicate with the kernel via system calls.

The `sh` is a Bourne shell and has the following basic functions:

- it takes input from a terminal or a file, interprets the input according to defined rules, and has the appropriate commands executed. A file containing input for the shell is called a shell procedure or shell script.
- it serves as a programming language. This means that you can create your own programs from existing Reliant UNIX commands and execute them without needing to compile them first.

Login shell

When you log in at a terminal, this shell is started up by default by `login(1)` . The login shell is associated with the calling terminal, i.e. the file descriptors 0 for standard input, 1 for standard output, and 2 for standard error output are specific to this terminal. The following steps are performed sequentially before the login shell displays its prompt:

- The login shell assigns default values to standard shell variables `HOME`, `IFS`, `MAIL`, `MAILCHECK`, `PATH`,

PS1, PS2 and SHELL (see **SHELL VARIABLES** below).

- The file /etc/profile is executed.
- The file $HOME/.profile is executed if you have created one.

You have to be in a login shell in order to log in with the Bourne shell built-in login(1), and even then it is only possible to log in as root.

You can find out if you are in a login shell as follows: The echo $0 command displays the invocation name of the current shell; -sh identifies a login shell, sh a normal subshell.

A login shell is started as a subshell whenever you log in to the system (again) with /bin/login (see **Subshells** below).

To exit from a login shell you press CTRL-D. If the login shell was not a subshell, the welcome screen is then displayed; your session has then terminated.

Subshells

You can use sh to call a new shell, known as a subshell. This shell is associated with the same terminal as its parent shell, and it inherits all shell variables that you have defined and exported in the parent shell and any shells above it [see export(1) and option -a]. Shell variables can only be passed from the current shell to subshells, not to the parent shell.

Unlike the login shell, the subshell does not execute /etc/profile or $HOME/.profile. This means that you can define or modify shell variables and switch directories in a subshell without affecting the parent shell. When the subshell exits, the parent shell returns in the state it was in when it started the subshell.

The Bourne shell built-in login(1) is rejected if you try to use it in a subshell. You have to be in a login shell if you want to use login.

The sh command allows you to control the behavior of the new subshell on the basis of the operands which you specify on the command line. If you give a file name, the subshell exits after executing the file. If you do not give a file name, the subshell is interactive: it reads input from the terminal until you terminate it by pressing CTRL-D. A shell running a shell script is not interactive (see also option -i).

OPTIONS

No operand specified:

A subshell is invoked with the -s option. The previously active shell process becomes the parent of the new shell. This new shell can be terminated with CTRL-D.

No option specified:

If you call sh with operands that are not options, *file* must be the first operand specified. *file* stands for the name of a shell script for which you will need to have read permission.

A subshell is then created. This subshell will execute the commands in the named shell script one after the other and then exit.

The *options* govern the behavior of the invoked subshell.

If you want to use more than one option, you have to group the code letters together with no spaces between them and with a single minus sign at the beginning.

The -- option cannot be combined with other options.

The -c *command* option, if selected, must be the last option specified, but the other options can appear in any order.

The behavior of the current shell can be controlled by using the Bourne shell built-in set(1).

There are two types of option:

- options that you can only specify with sh;
- options that you can also specify with the Bourne shell built-in set [also refer to set(1)].

Options for sh only

-c *command*

The subshell executes *command* and terminates. This option must be the last option you specify on the command line, and it must not be followed by any other operands either.

The `-c` *command* option allows you to invoke commands within a C program.

command is mandatory; omitting it produces an error message.

command

Must be a command line argument. If the *command* you want to run contains blanks or tabs, you have to enclose it in single or double quotes. Otherwise the shell will stop interpreting it at the first blank.

command is a command or executable shell script for which you will need to have read and write permission. It is run in the invoked (forked) subshell.

If you want to include a number of commands and/or shell scripts, you will have to escape the command separators.

If you use the basic file name (file basename) for the command/shell script, the shell looks for the file in the directories whose full path names are assigned to the `PATH` variable.

`-i` An interactive subshell is called. This shell reads input from standard input, processes it, starts the corresponding commands, and waits for them to terminate. Then it displays its prompt and is ready to process further input. An interactive shell responds to signal 2 for `SIGINT`, 3 for `SIGQUIT`, and 15 for `SIGTERM` as follows (*PID* is the process ID of the interactive shell):

Signal	Sent by	Interactive shell
`SIGINT`	DEL `kill -2` *PID*	Command running; ignored, i.e. an interactive shell is not terminated. □ Otherwise: issues new prompt.
`SIGQUIT`	CTRL-\ `kill -3` *PID*	Command running: ignored, i.e. an interactive shell is not terminated. □ Otherwise: issues a new prompt.
`SIGTERM`	`kill -15` *PID*	Ignored.

You can use the `-i` option to invoke an interactive shell within a C program.

`-i` not specified:

A shell is interactive by default if its standard input, standard output and standard error are associated with a terminal.

A shell running a shell script is not interactive. A script shell of this type responds to the `SIGINT`, `SIGQUIT`, and `SIGTERM` signals (sent by `kill -2`, `kill -3` and `kill -15`) as follows:

Signal	Script shell	Script shell &
`SIGINT`	waits for termination of executing command, then exits	ignored
`SIGQUIT`	waits for termination of executing command, then exits	ignored
`SIGTERM`	waits for termination of executing command, then exits	ignored

`-p` (privileged) The invoked shell does *not* set the effective user and group IDs to the real user and group IDs (those of the caller), but to it own user and group IDs.

`-r` Calls a restricted shell, in which the following restrictions apply:

- The Bourne shell built-in `cd(1)` is rejected, which means you cannot change your current directory.
- The value of the `PATH` variable cannot be modified.
- Commands are rejected if the command file name contains a slash. This means that you can only execute commands that are located in your current directory or in directories whose path names

have been assigned to the PATH variable.

- Commands are rejected if the command line contains the character > or >>. In other words, you cannot use > or >> to redirect the output of commands to a file.

A restricted shell can only be called with sh –r. The rsh command, by contrast, is used to invoke a shell on a remote system and not, as in earlier UNIX versions, a restricted shell.

For the system administrator only

As system administrator you can set up an environment for users who are only to be allowed to work in a restricted shell. For such users you define:

- The directory they are to work in.

  Users should not be able to work in the login directory, but should simply have read permission there.

- The commands they are allowed to use.

  You can do this by setting up a subdirectory in the login directory and copying the approved commands into it.

  Among other things, it should not be possible to invoke a subshell, as a subshell of a restricted shell is not restricted.

- The PATH variable.

In the user's $HOME/.profile you enter the appropriate commands and a call to the restricted shell. Once the login shell has executed the .profile, the user should only be able to work in the assigned working directory and in the restricted shell.

–r not specified:

The described restrictions do not apply to the invoked subshell.

–s The invoked subshell reads from standard input and writes on standard error. Bourne shell built-ins will, however, continue to write on the standard output.

The invoked subshell sequentially assigns the first nine arguments after –s to positional parameters $1 through $9. It assigns all command line arguments after –s collectively to the shell parameters $@ and $*; and it assigns the total number of arguments to the shell parameter $#. If there are fewer than nine arguments, the remaining positional parameters are set to the null string.

These values are only known to the called subshell. If positional parameters are to be set in the current shell, the Bourne shell built-in set(1) must be used instead.

If you also specify a *file* in the command line, the invoked subshell assigns the command line argument *file* to positional parameter $1, but does not execute the commands contained in *file*. You cannot set the –s option if you want to use sh to run a shell script.

–s not specified:

–s is set by default when sh is called without operands.

The login shell is always invoked with –s.

Options for sh and set

–a Automatically exports all shell variables that you define or redefine in the called subshell.

set +a turns off option –a in this subshell.

–a not specified:

Shell variables that you define or redefine in the called subshell are not automatically exported. In other words, a shell variable will not be known in a further subshell unless you explicitly export it using the Bourne shell built-in export(1) .

–e The invoked subshell or shell script exits immediately if a command exits with a non-zero exit status.

set +e turns off option –e in this subshell.

–e not specified:

If the called subshell is interactive, you can terminate it with CTRL-D. If it is a script shell, it only exits prematurely if it encounters a syntax error; otherwise, it does not exit until the last command has been executed, regardless of the exit status of individual commands.

`-f` Disables the file name generation mechanism in the called subshell or shell script. The symbols *, ? and [...] are not treated as metacharacters and expanded to generate all the matching file names.

`set +f` turns off option `-f` in this subshell.

`-f` not specified:

Matching file names are substituted for the above patterns.

`-h` This option changes the behavior of the called subshell when you define shell functions (see *Example 2*):

The commands you include in a shell function in the called subshell are entered in the hash table at the time of function definition [see `hash(1)`].

`set +h` turns off option `-h` in this subshell.

`-h` not specified:

Commands used in a shell function are not entered in the hash table until the function is executed.

`-k` Keyword. Variable assignments, i.e. arguments in the form *name*=*value*, are allowed at any point on the command line in the called subshell. The called subshell performs the assignment and makes the resultant keyword parameter available to the environment of the command.

`set +k` turns off option `-k` in this subshell.

`-k` not specified and not set previously:

Variable assignments must precede the command name. The shell makes the appropriate keyword parameters available to the environment of the command.

`-n` The called subshell reads and interprets all commands but does not execute them, thus omitting the last step in the processing of a command line (see **COMMAND LINE PROCESSING** below).

The `-n` option can be used to check a shell script for syntax errors (see *Example 1*).

Caution:

You cannot explicitly turn off option `-n`; it is turned off when the subshell exits.

`-n` not specified:

All commands are read, interpreted, and executed.

`-t` Terminate. The called subshell exits after executing the first command line.

If you invoke `sh -t` without a *file* operand, the subshell expects you to enter the command to be executed in the next line; but it does not issue a prompt.

`-t` not specified:

An interactive shell exits when you press CTRL-D.

A script shell only exits prematurely if it encounters a syntax error. Otherwise, it exits on reaching the end of the script.

Caution:

If the `-e` option is set, the shell exits immediately if a command returns a non- zero exit status.

`-u` Causes the called subshell to issue an error message on encountering an unset shell variable in a command line. The command in question is not executed.

Shell scripts are terminated as soon as the subshell encounters an undefined shell variable.

`set +u` turns off option `-u` in this subshell.

`-u` not specified:

The subshell does not issue an error message for an unset shell variable, but substitutes the null string as its value by default.

`-v` Verbose. The called subshell displays each subsequent input on standard error before executing the corresponding command.

In combination with the `-x` option, this option is useful for debugging shell scripts (see *Example 1*).

`set +v` turns off option `-v` in this subshell.

`-v` not specified:

The called subshell does not echo its input.

`-x` The called subshell interprets its input and writes it on standard error, marking each command that it processes with a plus sign (+) at the start of the line. A command consisting solely of a variable assignment is output without the plus sign. The shell then executes the command.

In contrast to the output with option `-v`, the `-x` option shows you how the shell has replaced metacharacters and shell variables.

Thus you can test a shell script as follows:

- either run the shell script with the command `sh -xv` *script*, or
- include `set -xv` as the first command in your shell script.

`set +x` turns off option `-x` in this subshell.

`-x` not specified:

The called subshell does not echo its input.

`--` This option cannot be combined with other options. The first argument specified after `--` is not interpreted as an option, even if it begins with a minus sign. This enables you to use `sh` to run a shell script which has a name beginning with a minus sign.

Note: In earlier versions of the shell, a single minus sign had the same effect that a double minus sign now has, namely to mark the end of the list of options. While a single minus sign can still be specified today, it does not have a recognizable effect.

file Name of the shell script to be executed by the subshell. You need read permission for this file.

If the `-c` *command* option is used, it must not be followed by any other operands, and no *file* may be specified.

If you invoke `sh` with the `-s` option, the invoked subshell simply assigns *file* to positional parameter `$1` and does not execute the commands contained in it.

file not specified:

The called subshell issues its prompt, reads input from standard input and processes it in accordance with the options which have been set.

argument

Any string delimited by blanks or tabs. The last argument is terminated by a command separator.

You can include any number of arguments, with at least one blank or tab between them.

The invoked subshell sequentially assigns the first nine arguments to positional parameters `$1` through `$9`. It assigns all command line arguments collectively to the shell parameters `$@` and `$*`; and it assigns the total number of arguments to the shell parameter `$#`. If there are fewer than nine arguments, the remaining positional parameters are set to the null string.

To access the tenth and subsequent command line arguments directly you need to use the `shift` command.

These values are only known to the called subshell. If positional parameters are to be set in the current shell, the Bourne shell built-in `set(1)` must be used instead.

argument not specified:

The null string is assigned as a value to positional parameters `$1` through `$9` and to the shell parameters `$@`, `$*`, and `$#`.

EXIT STATUS

`0` The shell returns a non-zero status when it detects a syntax error in the input. If the error occurs during execution of a shell script, the script file is terminated.

In all other cases, the exit status of the last command executed is returned.

FILES

`/etc/profile`
File executed by every login shell in order to set up a shell environment. This file is created by the system administrator.

`$HOME/.profile`
Shell script that users can set up in their HOME directory. `$HOME/.profile` is executed by the login shell after `/etc/profile`.

`/tmp/sh*`
Temporary files.

`/dev/null`
Null file to which standard input is directed. This file is typically used for background commands.

ENVIRONMENT VARIABLES

See **Standard shell variables** below.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In file name generation patterns, the `LC_COLLATE` environment variable governs the scope of character ranges, equivalence classes and collating elements, and the `LC_CTYPE` environment variable governs the scope of character classes. The latter also determines which characters from the currently valid character set are defined as letters.

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

Caution:

The current character set and the current collating sequence together determine which characters form part of a `[`*c1*–*c2*`]` range. For example, in the locale `LANG=LC_COLLATE=En_US.ASCII` the command `rm [a-z]*` deletes all files with names beginning with a lowercase letter from a to z inclusive. In a number of other locales the range a-z also include uppercase letters, and in these environments the same command would also delete files with names beginning with a capital letter. What is more, there are languages, Norwegian, for example, where some letters collate after z. So in an internationalized environment you should always represent character classes such as lowercase letters using expressions of the form `[:`*class*`:]`. Thus instead of the command given above you would use `rm [[:lower:]]*`.

Shell scripts written in a non-internationalized environment (or without allowing for the problems described above) should only be run in a non-internationalized environment; in other words, the NLS variables `LANG`, `LC_COLLATE` and so on should be undefined or null.

EXAMPLES

Example 1

This example searches for syntax errors in a newly created shell script without executing the commands in it. The shell script `syntax` has the following contents:

```
: # Invocation with sh syntax
set `echo *`
echo `pwd`:
```

```
ll {
ls –al $* | pg
}
```

The definition of the shell function `ll` contains an error; the parentheses () have been omitted. The shell reports this syntax error if you call `syntax` as follows:

```
$ sh –nv syntax
set `echo *`
echo `pwd`:
ll {
ls –al $* | pg
}
syntax: syntax error at line 6: `}'unexpected
```

Since the `–v` option has also been set, each read command line is additionally displayed by the shell. The individual commands are are not executed.

Example 2

The commands in a shell function are to be entered in the hash table at definition time rather than at runtime:

```
$ sh –h
$ ll() {
ls –al $* | pg
}
$ hash
hits cost command
0 2 /usr/bin/pg
0 1 /bin/ls
.
.
.
```

Option `–h` is set in the called subshell, so as soon as the `ls` and `pg` commands are defined in the shell function `ll` they are also entered in the hash table [see `hash(1)`].

THE SHELL AS COMMAND INTERPRETER

This section has the following layout:

- Entering and grouping commands
  - Pipelines
  - Command lists
  - Command grouping (bracketing)
  - Shell built-ins
- Shell metanotation
- Shell variables
  - Defining shell variables
  - Accessing the value of a variable
  - Standard shell variables
- The environment
  - Initial environment of a command
  - Runtime environment of the current shell
- Shell parameters
  - Substituting values for shell parameters

- Defining default values for shell parameters
- Command substitution
- Command input and output redirection
  - Redirecting standard input for a command
  - Redirecting standard output for a command
  - Redirecting standard error for a command
  - Multiple redirections
- File name generation
- Command line processing

Entering and grouping commands

Commands can either be collected in a file and executed as a shell script or entered interactively at the shell prompt. The default command format is:

*command*ERROR*parameter*ERROR...CR

command
: Name of a Reliant UNIX command, shell script, Bourne shell built-in or shell function to be executed.

parameter
: Command-line argument that the shell either interprets itself or passes on to *command*. Depending on the specified command, several command line arguments may be given.

The command name and the command line arguments are separated by tabs or blanks; a newline character terminates the last argument (thereby terminating command input). It is also possible to write a number of commands on the same line, with a command separator between them (see **Pipelines** and **Command lists** below).

Blanks, tabs, and newline characters are assigned as default argument separators to the `IFS` shell variable. The contents of the `IFS` variable are evaluated in the following cases:

- when the Bourne shell built-in `read(1)` parses the read input line into arguments,
- when the shell parses the modified command line into arguments for the second time (see "Step 8" in **COMMAND LINE PROCESSING** below).

If you modify the value of the `IFS` variable, `read` and the shell understand only your modified argument separators on the second parse run.

On the first parse run (see "Step 3" in **COMMAND LINE PROCESSING** below), blanks and tabs act as argument separators, regardless of the value of the `IFS` environment variable.

Every Reliant UNIX command returns its exit status to the shell in which it was called. The usual values of the exit status are:

`0` successful execution of the command

`0` if the command has failed

Interpreting single commands is, however, just one of the many facets of the shell. For instance, you can also string commands together as follows:

- Two or more commands can be linked by "pipes", represented by the vertical bar |, to form a pipeline (see **Pipelines** below).
- A series of commands or pipelines can be connected to form a command list by a semicolon (;) or by an ampersand (&) or by && or || (see **Command lists** below).

Pipelines

A pipeline is a sequence of two or more commands separated by the vertical bar character |, which forms a connection known as a pipe between two commands.

A pipe between two commands redirects standard output of the first command to standard input of the

second. Commands connected in this way must therefore satisfy the following conditions:

- The first command in a pipeline must write to standard output.
- The last command in a pipeline must read from standard input.
- If a pipeline comprises more than two commands, all intervening commands between the first and last must read from standard input and write to standard output. Commands that read from standard input and write to standard output are also known as filters.

The shell starts all commands of a pipeline simultaneously as independent parallel processes. On a single-processor system, however, only one of these processes can be handled by the processor in any one unit of processor time; the processor takes each process in turn.

The only process which does *not* receive its input from another process is the process associated with the first command in the pipeline.

The only process which does *not* pass its input on to another process is the process associated with the last command in the pipeline.

The individual processes must therefore be synchronized by the operating system kernel. The processes associated with two successive commands in the pipeline communicate with one another via a buffer. The reading process thus waits for the writing process to enter data in the buffer; and when the buffer is full the writing process waits for the reading process to extract the data from it.

The shell waits for the last command in the pipeline to complete and does not process further input until this has happened.

Exit status of a pipeline

The exit status of a pipeline is the exit status of its final command.

Command lists

A sequence of commands or pipelines can be combined to form a command list. The individual elements of such a list are separated by the ; or & character or the character pair && or ||. A command list may be terminated by a newline, semicolon, or ampersand. Any newline character not enclosed within parentheses or braces causes the shell to execute the command list (see **Command grouping** below).

This feature permits you to enter several commands at a time without waiting for individual commands to terminate. The shell does not issue its prompt until all commands in the command list have been executed.

In the following discussion, the term command encompasses both commands and pipelines. The meanings of individual command separators are given below:

Symbol	Effect
;	The shell processes the next command only after the preceding one finishes. The prompt does not appear between the two commands.
&	The shell runs the preceding command in the background and immediately processes the following command. Background processes ignore the `SIGINT` and `SIGQUIT` signals [see `trap(1)`]. □ The standard input of all asynchronously processed commands is redirected by the shell to the null file `/dev/null`. You can also redirect it to a different file, for example by including < *file* on the command line.
&& (ANDIF)	Causes the shell to execute the next command only if the preceding command has returned a zero (successful) exit status.
|| (ORIF)	Causes the shell to execute the next command only if the preceding command returned a non-zero (unsuccessful) exit status.

Precedence of command separators

The character pairs && and || have equal precedence, which is higher than the precedence of ; and &.

Colon ; and ampersand & have equal precedence.

Example:

command1&*command2*||*command3*;*command4* CR

The shell runs *command1* in the background and immediately processes *command2*. If *command2* returns an exit status of 0, the shell then executes *command4*.

If *command2* returns a non-zero exit status, *command3* and *command4* are then executed one after the other.

The || link overrides & and ;.

Exit status of a command list

The exit status of a command list is the exit status of the last command executed.

Command grouping

Parentheses and braces can both be used to group a number of commands. This mechanism allows you to

- run a group of commands in the background in a fixed sequence,
- collect the output of all the commands in one file or pipe it to another command.

(*commandlist*)

commandlist

List of commands or pipelines separated by command separators. The shell starts a subshell, which executes *commandlist* and then exits. The original shell returns, with the same shell variables and working directory as before.

If you redirect standard output to a file outside the parentheses, the output of all commands/pipelines in the list is combined and sequentially written to the specified file. This also applies if standard error output is redirected to a file.

If you put a pipe symbol | after the right parenthesis, the output of all commands in the list is passed as input to the command that follows the pipe.

If you put an ampersand & after the right parenthesis, a subshell is started in the background. This subshell sequentially executes all commands in the command list.

Example:

```
$ pwd;(cd /dev;pwd);pwd
/home/rose
/dev
/home/rose
```

In the above example, the invoked subshell makes a directory change to `/dev`. However, this change in the current directory has no influence on the parent shell.

Exit status

The exit status is the value returned by the last command executed in *commandlist*.

{ *commandlist*; }

commandlist

List of commands or pipelines separated by command separators.

The current shell executes *commandlist*. The braces are shell keywords and thus must each be preceded by a command separator and followed by an argument separator.

In the following cases, however, *commandlist* is executed in a subshell despite the braces, i.e. the braces have the same effect as parentheses:

- If you redirect standard output to a file outside the braces.

  In this case the output of all commands in the list is combined and sequentially written to the specified file.

  This also applies if standard error output is redirected to a file.

- If you put a pipe symbol | after the right brace.
  In this case the output of all commands in the list is passed as input to the command that follows the pipe.
- If you put an ampersand & after the right brace.
  A subshell is started in the background and sequentially executes all commands in the command list.

Examples

Example 1

If no `.profile` file exists, a message to that effect is to be displayed and the current shell is to exit:

```
[ -f .profile ] || {
echo No .profile created
exit 1
}
```

As the commands within the braces are executed by the current shell, `exit` terminates the current shell. The || separator ensures that the list in the braces is only executed if the Bourne shell built-in `test(1)` returns a non-zero exit status.

Example 2

If standard output of the command list is redirected to a file, the command list is executed in a subshell in spite of the braces, as if it were in parentheses:

```
$ pwd;{ cd /dev; echo Terminals:; ls tty*;} > $HOME/output
/usr1/rose
$ pwd
/usr1/rose
$ cat output
Terminals:
tty000
tty001
tty003
tty010
```

The current working directory does not change, because the command list is executed in a subshell. The shell combines the output of the `echo` and `ls` commands and writes it to the file `$HOME/output`.

Exit status

The exit status of the *commandlist* is that of the last command executed.

BOURNE SHELL BUILT-INS

Some commands are executed by the shell itself and hence do not require a new process to be started. Such commands are known as built-in commands of the Bourne shell (or Bourne shell built-ins) and differ from the other external commands in the following respects:

- They constitute subroutines of the binary file `/bin/sh` and thus have predefined names which cannot be changed. You can change the access permissions for `/bin/sh`, but not for the shell's individual built-in commands.
- The shell's built-in commands are given precedence over external commands (see **COMMAND LINE PROCESSING** below). If there is an external command with the same name as a shell built-in, the shell always executes the built-in unless you call the external command by its absolute path name (e.g. `/bin/pwd`).
- The built-in commands are faster because the shell executes them itself.

There are no external equivalents to most of the Bourne shell built-ins. In fact, it is not even possible to have one for the built-in command `cd`: if a new process were to be created to execute `cd`, the current working directory would change only for the duration of that process. On completion of the command, the process would also be completed; the calling shell would return, and no change of directory would have taken place.

The output format of Bourne shell built-ins is not standardized, so you should not rely on a particular output format when using them in programs or shell scripts.

Since most of the built-in commands of the Bourne shell are described in the normal alphabetical sequence (see **SEE ALSO** below) along with the external commands, they are simply listed briefly here: □

Bourne shell built-in	Function	External equivalent
`bg`	move a job into the background	
`break`	exit from the enclosing loop	
`cd`	change working directory	
`chdir`	change working directory	
`continue`	resume execution at the next iteration of the enclosing loop	
`echo`	echo command line arguments	`/bin/echo`

Bourne shell built-in	Function	External equivalent
`eval`	evaluate command line arguments and execute as command	
`exec`	overlay current shell	
`exit`	terminate shell script	
`export`	export shell variables	
`fg`	move job into the foreground	
`getopts`	check arguments for legal options	`/bin/getopts`
`hash`	process shell hash table	
`jobs`	list information about a job	
`kill`	send/list signals	`/bin/kill`
`newgrp`	change to new group	`/bin/newgrp`
`pwd`	print working directory	`/bin/pwd`
`read`	read arguments from standard input and assign to shell variables	
`readonly`	protect shell variables	
`return`	causes a shell function to return to the invoking script	
`set`	set shell options or positional parameters	`/bin/sh`
`shift`	shift values of positional parameters to left	
`stop`	`stop` *PID* sends a signal (= `kill -23` *PID*)	
`suspend`	send a signal by a user (= `kill -24` or CTRL-Z)	
`test`	test conditions	`/bin/test`
`times`	print accumulated times for processes run from the shell	

`trap`	define signal handling	
`type`	display type of command	
`ulimit`	set or display user file size limits	
`umask`	set file-creation mode mask	
`unset`	delete shell variables/shell functions from the environment	
`wait`	wait for background processes to terminate	
`:`	return exit status 0, see `colon(1)`	`/bin/true`
`.`	execute shell scripts in the current shell, see `dot(1)`	
`[`	test conditions, see `test(1)`	

SHELL METANOTATIONS

This section contains a list of characters and character combinations that have a special meaning for the shell. The table below briefly indicates the special meaning of each metacharacter and gives a reference to the section in which it is discussed in detail [see also `specialchar(5)`].

The metacharacters contained in a command line are evaluated by the shell in several steps (see **COMMAND LINE PROCESSING** below). □

Metacharacter	Meaning	Described in
blank newline tab	argument separator	Entering and grouping commands
newline | ; & | | &&	end of command or command separator	Command lists
(*commandlist*) { *commandlist*;}	grouping of commands	Command grouping
`*command*` * ? [*s*] [*c1–c2*] [!*s*] [!*c1–c2*]	replace by output replace with any matching file names	Command substitution File name generation
>*file* >>*file* >&*number*	redirect standard output close standard output	Command input and output redirection

>&–		
<*file* <<*argument* <<–*argument*	redirect standard input	Command input and output redirection
<&*number*	close standard input	
<&–		

Metacharacter	Meaning	Described in
name=*value*	assign value	Shell variables
$*name* ${*name*}	access value of variable	Shell parameters
${*name*–*value*} ${*name*?*value*} ${*name*+*value*}	replace on condition that variable is defined	Defining default values for shell parameters
${*name*=*value*}	assign if variable is not defined	Defining default values for shell parameters □
${*name*:=*value*}	assign if variable is not defined or is set to the null string	
${*name*:–*value*} ${*name*:?*value*} ${*name*:+*value*}	replace on condition that variable is defined and is not set to the null string	Defining default values for shell parameters
$0	1st command line argument	Shell parameters
$1, $2, ... , $9	positional parameters	
$*	all command line arguments	
"$*"	as `$*` but as one argument	
$@	all command line arguments	
"$@"	as `$@` but as `$#` arguments	
$#	number of command line arguments	
$$ $! $? $–	read-only shell parameters	Shell parameters
name() { *commandlist*;}	shell function	Shell functions
\ '...' "..."	escape characters	Quoting shell metacharacters
#	comment character	Comments in shell scripts
;;	command list terminator in `case`	Shell control-flow statements

	constructs	

Quoting shell metacharacters

Every shell metacharacter can be escaped, i.e. made to represent itself. Depending on the character you wish to escape (or quote), one of the escape characters described below can be used.

The escape characters themselves do not form part of the command line arguments. The shell strips these characters before passing the arguments on to the command.

Escape characters

\ Escapes the special meaning of the next character. The backslash \ can be used to escape all shell metacharacters.

\ CR causes the shell to ignore the special meaning of the newline as an argument or command separator. The shell ignores this entry and strips both the backslash and the newline character from the command line.

'*string*'

Escapes the special meaning of all characters in *string*. Single quotes can be used to escape all shell metacharacters with the exception of a single quote itself.

Since blanks, tabs and newline characters enclosed within single quotes lose their special meaning as argument separators, the shell treats the entire *string* as a single argument.

"*string*"

Only the following characters retain their special meaning for the shell:

$ when followed by a variable name is replaced by the value of the shell variable.

'*command*'

is replaced by the output of *command* (see **COMMAND SUBSTITUTION** below).

\ escapes the special meaning of the metacharacter following it.

Only the characters $, ` ...` , and \ retain their special meaning within double quotes. If none of these metacharacters follows the backslash, the backslash is treated as a character that is part of the string.

All other metacharacters within the double quotes are treated as being escaped.

Since blanks, tabs and newline characters enclosed within double quotes lose their special meaning as argument separators, the shell treats the entire string as a single argument. There is, however, an exception here: "$@" is the same as "$1" "$2" ... "$9", i.e. the individual command line arguments are retained as separate arguments.

Effect of single and double quotes (and backquotes)

escape	argument	command	\	$	*	?	[	`	"	'
	□□□□□□□□□separator									
'...'	yes	yes	yes	yes	yes	yes	yes	yes	yes	*e*
"..."	yes	yes	no	no	yes	yes	yes	no	*e*	yes
`...`	yes	yes	yes	no	no	no	no	*e*	no	no

e stands for end of quoted string

SHELL VARIABLES

Shell variables are always of the type string. They do not need to be declared; they are held to be defined as soon as you assign a value to them.

Shell variables are useful when you wish to store strings that are frequently used. Storage space requirements are taken care of by the shell itself.

In the following section you will learn

- how to define and assign a value to a variable
- how to access the value of the variable
- the standard variables used by the shell.

Defining shell variables

Every shell variable consists of a name and a value and is defined as follows:

name=*value* [*name*=*value*] ...

name Name of the variable. A variable name must begin with a letter or an underscore (_). The first character may be followed by letters, digits, and underscores in any order.

A shell variable and a shell function cannot be given the same name in the current shell.

If a variable with the specified name has already been defined, the new assignment causes its original value to be forgotten in the current shell.

= The assignment operator. There should be no blanks between *name* and *value*.

value Value of the variable.

If *value* contains blanks, tabs, or newline characters, they must be escaped. The shell assigns the characters *, $, and [...] to the *name* variable without interpreting them. *value* is thus not replaced by all matching file names (see **FILE NAME GENERATION** below).

If you omit *value* or specify "" or ''for it, the null string will be assigned to *name*.

The assignment can either be made interactively or included in a shell script. You can also write several assignments on one line, separated by the usual argument separators.

Variables defined in this way are by default only known in the current shell. The Bourne shell built-in `export(1)` must be used if they are also to be recognized in every subshell. Variables can be exported down to subshells but not up to parent shells. See **THE ENVIRONMENT** below for further details.

The Bourne shell built-in `read(1)` can also be used to define shell variables.

The Bourne shell built-in `unset(1)` removes the definition of a shell variable.

Accessing the value of a variable

The Bourne shell built-in `set(1)` lists all variables defined in the current shell along with their values. You can, however, also directly access the value of a variable as follows:

$*name*

name Name of the shell variable whose value you want to access. The name must come immediately after the dollar sign without intervening space.

name is a shell variable, while $*name* represents a keyword parameter. You can assign a value to a variable, but not to a keyword parameter (see also **SHELL PARAMETERS** below).

Example:

To find out the value of the `HOME` variable, you enter:

```
$ echo $HOME
```

The Bourne shell built-in `echo(1)` writes the value of the `HOME` variable on standard output.

Standard shell variables

Some standard variables are automatically passed to the login shell when it is started. Before displaying its first prompt, this shell defines a few additional standard variables itself.

Most standard variables are assigned a default value, which is also known in every subshell. It is not necessary to `export` the values of standard variables to subshells unless you change them from the default. Otherwise the default values automatically apply in every subshell.

If you wish to define shell variables permanently, you can enter the appropriate assignments in your `$HOME/.profile` file. This file is executed by the login shell before it issues its prompt.

The following standard variables are recognized by the shell:

CDPATH=*directory*[:*directory* ...]

The path names of directories to be searched by the cd command can be assigned to the CDPATH variable. The individual path names must be separated from one another by a colon. You can identify the current directory by entering a dot (.) or the null string. A colon immediately after the assignment operator assigns the working directory as the first directory in the list.

The Bourne shell built-in cd(1) accesses CDPATH when searching for the directory that is to be made the working directory. The search begins in the directory whose path name precedes the first colon. All path names contained in the CDPATH variable are processed from left to right [see cd(1)].

Default value: none, i.e. by default CDPATH is undefined.

If CDPATH is undefined, cd looks for relative path names relative to the current working directory.

HOME=*directory*

The HOME variable is assigned the absolute path name of the home directory. If you invoke the Bourne shell built-in cd(1) without arguments, cd accesses HOME, and the home directory becomes the new working directory.

The login shell looks for the .profile file in the directory assigned to the HOME variable.

Default value: The login command assigns the absolute path name of your login directory as the value of the HOME variable. This path name is allocated to your login name in the /etc/passwd file.

IFS=*c*...

The values of the IFS variable are argument separators, where *c* is any single character. Blanks, tabs, and newline characters have to be escaped. A list of characters has to be entered without intervening blanks.

The shell evaluates the content of the IFS variable in the following cases:

- when the Bourne shell built-in read(1) parses an input line into arguments
- when the shell does its second parse run on the command line (see **COMMAND LINE PROCESSING** below, Step 8).

The IFS variable cannot be removed from the environment with unset(1) .

Default value: blank tab newline. The login shell assigns this value to IFS.

If you modify the value of the IFS variable, read and the shell on its second parse run will only understand the modified argument separators.

LANG=*language*

The NLS variable LANG is used to define an internationalized shell environment. The value of LANG is the name of an existing database. These databases reside in the directory /usr/lib/locale. If the LC_ALL variable is not set, LANG defines not only the native language but also everything that can be defined individually in an internationalized program with the NLS variables LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME and LC_MESSAGES.

If LANG is undefined or null or has an invalid value, the system acts as if it were not internationalized.

LC_ALL=*language*

The value of LC_ALL overrides all LC_* variables and LANG.

LC_COLLATE=*language*

If the LC_ALL variable is not set, the NLS variable LC_COLLATE defines the collating sequence for ordering strings of characters.

If LC_COLLATE is undefined or is assigned the null string, its value defaults to the value of LANG. If LC_COLLATE has an invalid value, an internationalized program will act as if it were not internationalized.

LC_CTYPE=*language*

If the LC_ALL variable is not set, the NLS variable LC_TYPE defines which character class each character belongs to, and how uppercase letters are converted to lowercase and vice versa.

If LC_TYPE is undefined or is assigned the null string, its value defaults to the value of LANG. If LC_TYPE has an invalid value, an internationalized program will act as if it were not internationalized.

LC_MESSAGES=*language*

If the LC_ALL variable is not set, the NLS variable LC_MESSAGES defines the language in which messages are displayed by an internationalized program. If LC_MESSAGES has an invalid value, an internationalized program will act as if it were not internationalized.

LC_MONETARY=*language*

If the LC_ALL variable is not set, the NLS variable LC_MONETARY governs currency symbols and monetary value formats.

If LC_MONETARY is undefined or is assigned the null string, its value defaults to the value of LANG. If LC_MONETARY has an invalid value, an internationalized program will act as if it were not internationalized.

LC_NUMERIC=*language*

If the LC_ALL variable is not set, the NLS variable LC_NUMERIC defines the radix character (decimal point), the exponentiation symbol and the thousands separator.

If LC_NUMERIC is undefined or is assigned the null string, its value defaults to the value of LANG. If LC_NUMERIC has an invalid value, an internationalized program will act as if it were not internationalized.

LC_TIME=*language*

If the LC_ALL variable is not set, the NLS variable LC_TIME defines the format for date and time strings.

If LC_TIME is undefined or is assigned the null string, its value defaults to the value of LANG. If LC_TIME has an invalid value, an internationalized program will act as if it were not internationalized.

LOGNAME=*loginname*

The login command assigns the LOGNAME variable the name under which you logged in on the system.

MAIL=*file*

The MAIL variable is assigned the absolute path name of the file to which mail writes incoming mail.

If the MAILPATH variable has not been defined, the shell informs you when new mail messages arrive in this file by issuing the message:

you have mail

Default value: /var/mail/$USER. The login command assigns this value to MAIL.

MAILCHECK=*integer*

The MAILCHECK variable is assigned an *integer* value. The shell accesses the MAILCHECK variable: every *integer* seconds it checks whether mail messages have arrived in your $MAIL file or in any of the files whose path names are assigned to the MAILPATH variable.

If you assign MAILCHECK a value of 0, the shell inspects your mail files whenever it is about to issue its prompt.

The following message is issued if mail has arrived: you have mail

The MAILCHECK variable cannot be removed from the environment with unset.

Default value: 600, i.e. the shell checks every 10 minutes (600 seconds) whether new mail has arrived in any of the files whose path names have been assigned to the MAIL or MAILPATH variables.

The login shell assigns this value to MAILCHECK.

MAILPATH=*file*[%*text*][:*file*[%*text*]]...

This variable can be used to specify the path names of files to be checked for new mail. The individual path names must be separated by a colon and represent files for which you have read permission.

In addition, each path name can be followed with a % sign and a *text* which is to be output by the shell when new mail arrives in the corresponding file.

The shell accesses the MAILPATH variable to determine whether new mail has arrived in any of the files

specified in this variable.

Default value: none, i.e. by default `MAILPATH` is undefined.

`PATH=`*directory*`[:`*directory*`]...`

The `PATH` variable is assigned the absolute path names of the directories in which the shell is to look for commands. A colon is used to separate the individual path names.

You can identify the working directory by entering a dot (.) or the null string, i.e. nothing.

For security reasons, the current working directory should always come last in the list of directories assigned to the `PATH` variable.

The shell accesses the `PATH` variable when searching for commands invoked under their file basenames, i.e. a name not including any slashes. The search begins in the directory whose path name precedes the first colon. All path names assigned to the `PATH` variable are processed from left to right until the shell finds the required file or establishes that the file does not appear in any of the listed directories.

The `PATH` variable cannot be removed from the environment with `unset`.

`PATH` has different default settings for the superuser and for non-privileged users. The appropriate value is derived from the entries in `/etc/default/login`.

The login shell assigns this default value to `PATH`.

`PS1=`*promptstring*

The `PS1` variable is assigned the prompt string that the shell is to issue when it is ready to accept input. Any blanks or tabs appearing in *promptstring* have to be escaped.

The `PS1` variable cannot be removed from the environment with `unset`.

Default value: `$`

The login shell assigns this default value to `PS1`.

`PS2=`*promptstring*

The `PS2` variable is assigned the *promptstring* that the shell is to issue when it expects further input after a newline character or when you have escaped a newline character. Any blanks or tabs appearing in *promptstring* have to be escaped.

The `PS2` variable cannot be removed from the environment with `unset`.

Default value: `>`

The login shell assigns this default value to `PS2`.

`SHELL=`*commandinterpreter*

The `SHELL` variable is assigned the absolute path name of the command interpreter. Whenever a subshell is invoked, the calling shell looks to see if the value of `SHELL` contains the letter `r`. If it does, the called subshell is a restricted shell.

The editors `ed` and `vi` access the `SHELL` variable if you escape to the shell or invoke a shell command while working with them.

Default value: The `login` command assigns this variable the absolute path name of the program that the system runs for you when you log in. The path name of this startup file is allocated to your login name in the `/etc/passwd` file and is `/bin/sh` by default.

`TERM=`*type*

The special process `getty` assigns the *type* of the terminal to the `TERM` variable. *type* is the corresponding entry in the file `/etc/gettytab`.

Editors access the `TERM` variable to find out the capabilities of the terminal.

`TZ=`*stdoffset1*`[`*dst*`[`*offset2*`][;`*start*`[/`*time*`],`*end*`[/`*time*`]]]`

`TZ=`*stdoffset1*`[`*dst*`[`*offset2*`][;`*start*`[/`*time*`],`*end*`[/`*time*`]]]`

`TZ=:`*zonename*

The `TZ` variable contains timezone information. The `date` command and most of the library functions which

reference the time of day use `TZ` to determine the timezone and to convert universal time UTC (Universal Time Coordinated) to local time and vice versa.

The individual components of the value of `TZ` follow one another without intervening blanks, e.g. MET-1 (Format 1 or 2) or :US/Pacific (Format 3). The meaning of each component is as follows:

Format 1:

std Standard local timezone. *std* consists of at least three letters, either uppercase or lowercase.

Note: *std* is not used in calculating the local time (this being the job of *offset1*). *std* simply defines the name of the timezone.

offset1

Indicates the value that must be added to the local time in order to arrive at Coordinated Universal Time (UTC, also known as Greenwich Mean Time, or GMT).

offset1 has the following format:

[]*hh*[:*mm*[:*ss*]]

If the specified time is preceded by a plus sign, the timezone is west of the Prime Meridian. If it is preceded by a minus sign, the timezone is east of the Prime Meridian.

not specified:

Defaults to +.

hh Hours (0-24)

mm Minutes (0-59)

ss Seconds (0-59)

Example:

Central European time (MET) is equal to UTC + 1 hour: `MET=UTC+1`, so `UTC=MET-1`. In this case *offset1* would be `-1`.

dst Summer timezone (daylight saving time; same format as *std*). As with *std*, *dst* simply defines the name of the summer timezone.

dst not specified:

Conversion to summer time does not apply.

offset2

Indicates the value that must be added to the local summer time in order to arrive at Coordinated Universal Time (UTC).

offset2 has the same format as *offset1*.

offset2 not specified:

summer time = standard time + 1 hour

start[/*time*],*end*[/*time*]

start[/*time*] defines when to change from standard time to summer time; *end*[/*time*] specifies when to change back. *start* and *end* specify the day on which the change is made; *time* indicates when this happens (in current local time).

start and *end* are given in the form of a number *n* (0 £ *n* £ 365), meaning the *n*th day of the Julian calendar. Leap days are counted, so in this format it is possible to refer to February 29.

time has the format: *hh*[:*mm*[:*ss*]]

(see *offset1*)

time not specified:

time = 02:00:00

start[/*time*],*end*[/*time*] not specified:

In Central Europe, the change from standard time to summer time occurs on the last Sunday in March; in the USA and Canada on the second Sunday in April. Summer time is changed back to standard time

on the last Sunday in October (at 02:00:00 in both cases).

Format 2:

Format 2 is like Format 1 with the addition of more flexible syntax for specifying when summer time starts and ends. Note that the only difference in the syntax synopsis is that Format 2 has a comma instead of a semicolon before *start*[/*time*],*end*[/*time*].

std See Format 1

offset1
See Format 1

dst See Format 1

offset2
See Format 1

start[/*time*],*end*[/*time*]
start[/*time*] defines when to change from standard time to summer time; *end*[/*time*] specifies when to change back. *start* and *end* specify the day on which the change is made; *time* indicates when this happens (in current local time).

start and *end* can have any of the following formats:

`J`*n* stands for the *n*th day of the Julian calendar, where 1 £ *n* £ 365. Leap days are not counted. In other words, February 28 is always day 59, and March 1 is always day 60. It is impossible to refer to February 29 in this format.

n stands for the *n*th day of the zero-based Julian calendar, where 0 £ *n* £ 365. Leap days are counted, and it is possible to refer to February 29.

`M`*m*.*w*.*d*
stands for the *d*th day of week *w* of month *m* (0 £ *d* £ 6, 1 £ *w* £ 5, 1 £ *m* £ 12).

Day zero of a week is Sunday. Week 1 of a month is the week in which the first day of the month falls. When a value of 5 is specified for a week (`M`*m*.`5`.*d*), the "last *d*-day in month *m*" is implied. For example, M9.5.6 refers to the last Saturday in September.

time has the format: *hh*[:*mm*[:*ss*]]

(see *offset1*)

time not specified:

time = 02:00:00

start[/*time*],*end*[/*time*] not specified:

In Central Europe, the change from standard time to summer time occurs on the last Sunday in March; in the USA and Canada on the second Sunday in April. Summer time is changed back to standard time on the last Sunday in October (at 02:00:00 in both cases).

Example:

In Central Europe, the clocks go forward on the last Sunday in March, and back again on the last Sunday in October, in both cases at 2 a.m. Thus the appropriate setting for the `TZ` variable is:

```
TZ=MET-1MDT,M3.5.0/02:00:00,M9.5.0/02:00:00
```

Format 3:

zonename
zonename is the name of a predefined timezone with a predefined change to summer time.

The accepted values for *zonename* are the names of the files under `/usr/lib/locale/TZ`. These files are created with the system administration command `zic(1M)` .

Examples:

In `/usr/lib/locale/TZ` there are files called `GMT` and `Greenwich-1` and a directory `US` including files called `Pacific` and `Yukon`.

```
$ TZ=:GMT date
Wed Oct 23 15:18:47 GMT 1991
$ TZ=Greenwich-1 date
Wed Oct 23 14:18:47 GMT-1 1991
$ TZ=:US/Pacific date
Wed Oct 23 08:18:47 PDT 1991
```

Note: Syntax errors in an environment variable such as `TZ` are not allowed to cause a command like `date` to abort; they have to trigger a default response instead. The appropriate default response for the environment variable `TZ` is as follows:

If *std* or *offset1* is unidentifiable (Format 1 or 2), or if the named file does not exist under `/usr/lib/locale/TZ` (Format 3), the clock reverts to GMT.

In other cases, only the valid part of the variable is interpreted, and the rest is ignored.

Examples:

```
$ TZ=invalid date
Wed Oct 23 15:27:02 GMT 1991
$ TZ=MET-1MDT,and_this_is_the_rest
Wed Oct 23 16:27:02 MET 1991
```

`USER=`*loginname*

The `login` command assigns the `USER` variable the name under which you logged in on the system. The `USER` variable has only been retained for compatibility with earlier versions; it has now been replaced by the `LOGNAME` variable, which serves the same purpose.

THE ENVIRONMENT

Whenever a process, including the shell, is started, it is provided with a range of information by the calling process. This includes:

- information that is essential for the execution of the process, i.e. information on parent and child processes, open files, the current directory, defined signal handling, etc.
- information on specific variables and the values assigned to them.

This information all forms part of the process environment. While a process is executing it may add further local variables to its environment or make local changes to the values of existing variables.

In the following section we explain:

- which variables the shell passes to the initial environment of a command (see **Initial environment for a command** below)
- how to modify the environment of the current shell (see **Runtime environment of the current shell** below).

Initial environment for a command

When you invoke a command, the shell passes to this command:

- the names and values of all shell variables that you have exported in the current shell or in a parent shell and have not removed from the environment with the `unset` command, and
- any standard variables whose values you have not changed or removed with `unset`. The standard variables `IFS`, `MAILCHECK`, `PATH`, `PS1` and `PS2` cannot be removed in any case.

These variables and their values represent the initial environment that is inherited by the invoked command from the shell that calls it. Use the `env` command if you wish to check which variables are passed by the current shell to each invoked command.

Augmenting the initial environment for a command

When you invoke a command, you can augment its environment without affecting the environment of the current shell or the initial environment of other programs executed by the shell. You do this by writing the appropriate assignments on the command line before the command name.

These assignments are not carried out in the current shell, but in the process that the shell starts in order to execute this command. The assigned variables are added to the environment of the command, and the command is therefore able to access them; but they are not available in the the current shell.

If the `-k` option is set in the current shell [see `set(1)`], such assignments can be written at any position within the command line. Variables thus defined are known only to the command invoked on that line.

Example:

The following two command lines lead to the same result:

```
customer=mason script
(customer=mason;export customer;script)
```

The first line contains an assignment before the command name. This assignment is not carried out in the current shell, but in the subshell that runs `script`. The `customer` variable is then passed to the script as part of its environment. On completion of the script, the subshell terminates, so the `customer` variable will not be known in the parent shell.

The second line is enclosed within parentheses, which means that all parenthesized commands are to be executed in a subshell. The subshell adds the `customer` variable to the environment for `script` and passes this environment, including the `customer` variable, to the script. The subshell terminates on reaching the end of the script, so the `customer` variable will not be known in the parent shell.

If you define variables in the current shell and export them using `export`, you modify both the environment of the current shell and the initial environment of any commands that are invoked subsequently (see next section).

Runtime environment of the current shell

When a shell is invoked, it first scans the environment that is passed to it with the call. For each variable pair it finds, it creates a shell parameter with the specified name and associated value.

The login shell started by `login` inherits its initial environment from `login`. The login shell itself initializes the variables `IFS`, `MAILCHECK`, `PATH`, `PS1` and `PS2`.

Every variable that you define or redefine in a shell is added as a local variable to the runtime environment of this shell. Local variables do not affect the environment of the command that starts the shell, however. You can verify this with the `env(1)` command.

The Bourne shell built-in `set(1)` prints all variables defined in the current shell, including all locally applicable variables.

You can change the runtime environment of the current shell together with the initial environment of all commands invoked subsequently:

- The initial environment for commands can be augmented with the aid of the Bourne shell built-in `export(1)`. Exported shell variables are added to the initial environment and are known to the current shell, all its subshells, and any commands that it invokes.
- A variable can be removed from the initial environment with the aid of the Bourne shell built-in `unset(1)`. The exceptions to this are the standard variables `IFS`, `MAILCHECK`, `PATH`, `PS1` and `PS2`, which cannot be removed.

  A deleted variable is also no longer available in the runtime environment of the current shell.

If the `-a` option is set in the current environment, all shell variables that you define or redefine are exported automatically [see `set(1)`].

It is also possible to modify the initial environment for a single command (see **Augmenting the initial environment for a command** above). In this case the shell's runtime environment does not change.

SHELL PARAMETERS

Shell parameters are prefixed by a dollar sign (`$`). The shell replaces a parameter on the command line with

the value currently assigned to the corresponding shell variable. Shell parameters are addressed as follows:

$parameter

parameter

Name of a shell variable or any of the following characters:

```
0 1 2 3 4 5 6 7 8 9 * @ # - ? $ !
```

If the name of a shell parameter is followed by a letter, digit or underscore, the shell interprets this character as part of the name, so the name has to be kept separate from any characters that follow it. One way of doing this is to put the name in braces:

${parameter}

Example:

```
$ file=/usr1/rose/tables/list1
$ mv $file ${file}00
```

Omitting the braces causes the shell to search for a variable named `file00`, which may well be undefined.

There are three types of parameter:

- keyword parameters,
- positional parameters and
- special shell parameters.

Keyword parameters

Keyword parameters are parameters in the form *$name* or *${name}*. The shell replaces a keyword parameter with the value of the corresponding shell variable. If this variable is not defined, the shell substitutes the null string for the keyword parameter.

You can create new keyword parameters or modify the value of old ones by defining or redefining the corresponding shell variable. You do this with the following assignment:

name=value

Here *name* is a shell variable, while *$name* is a shell parameter. Whereas a variable can be assigned a value, a shell parameter cannot (see the sections **SHELL VARIABLES** and **THE ENVIRONMENT** above).

Keyword parameters may be passed to every invoked subshell by including the corresponding variable in the environment with the shell built-in `export(1)`.

Positional parameters

Positional parameters are the parameters `$1`, `$2`, `$3`, `$4`, `$5`, `$6`, `$7`, `$8` and `$9`. The following command line arguments are substituted for these positional parameters by the shell:

`$1` first command line argument

.

.

.

`$9` ninth command line argument

You can thus access the first nine command line arguments directly, and use `$*` or `$@` to access all the command line arguments collectively. By using the Bourne shell built-in `shift(1)`, you can access the tenth and subsequent arguments directly as well.

In an interactive shell, the only way to assign values to all positional parameters at once is indirectly, with the Bourne shell built-in `set(1)`. Following the invocation of `set`, these values may also be accessed in the current shell.

The `sh -s` command can be used to indirectly assign values to all positional parameters in the invoked subshell at once.

Unlike keyword parameters, positional parameters cannot be directly redefined, and their values cannot be

exported to the environment by `export`.

Special shell parameters

The special shell parameters and their values:

`$0` Invocation name of the command or the shell script; with shell functions, $0 is the name of the current shell or the name of the shell script, depending on where you called the function.

`$*` All command line arguments, i.e. positional parameters $1 through $9 plus any other command line arguments.

`"$*"` all command line arguments as one argument, i.e. "$1□...□$9□..."

`$@` All command line arguments, i.e. positional parameters $1 through $9 plus any other command line arguments.

`"$@"` all command line arguments as separate arguments, i.e. "$1"□...□"$9"□...

`$#` The total number of command line arguments; this value may be greater than 9.

`$-` All options that have been set in the current shell.

`$?` The exit status of the last executed command; this value does not apply to background commands.

`$$` The process ID (PID) of the current shell

`$!` The process ID (PID) of the last command started in the background.

These values are assigned by the shell automatically. The parameters `$*`, `$@` and `$#` are modified when you assign new values to positional parameters by using the `set` or `sh –s` command or when you use `shift`.

Substituting values for shell parameters

When processing the command line, the shell substitutes values for the shell parameters on the command line (see **COMMAND LINE PROCESSING**, Step 4). The value of a shell parameter can be accessed as follows:

$*parameter* or ${*parameter*}

If no value has been defined, the shell replaces the specified parameter with the null string.

Defining default values for shell parameters

If you want to ensure that a shell parameter is always replaced by a predefined value, you can define a default value for it. The condition under which the shell is to substitute this default value can be specified together with the definition.

This section describes how you can define such default values. The value for *parameter* may be given as the name of a shell variable or any of the following characters:

`0 1 2 3 4 5 6 7 8 9 * @ # - ? $ !`

The meanings of these characters are explained in the section on **SHELL PARAMETERS** above.

Any arbitrary strings are permitted for the specified *default_value*. This *default_value* is only interpreted by the shell when it replaces the corresponding parameter with this value (see the example at the end of this section).

The following conventions are recognized by the shell:

${*parameter–default_value*}

The shell checks that *parameter* is defined, i.e. already assigned a value (note that the assigned value may be the null string).

If it is, it substitutes the defined value.

If it is not, it substitutes the *default_value*.

This is not equivalent to assigning a value to the parameter: the *default_value* applies for this access only.

${*parameter*:-*default_value*}

The shell checks that *parameter* is defined, i.e. already assigned a value (but not the null string; the colon precludes the null string).

If it is, it substitutes the defined value.

If it is not, it substitutes the *default_value*.

This is not equivalent to assigning a value to the parameter: the *default_value* applies for this access only.

${*parameter*=*default_value*}

parameter must be the name of a shell variable.

The shell checks that *parameter* is defined, i.e. already assigned a value (note that the assigned value may be the null string).

If it is, it substitutes the defined value.

If it is not, it assigns the *default_value* to *parameter* and substitutes its value.

Only shell variables can be assigned values in this way, so $*parameter* cannot be a positional parameter or a special shell parameter.

${*parameter*:=*default_value*}

parameter must be the name of a shell variable.

The shell checks that *parameter* is defined, i.e. already assigned a value (but not the null string; the colon precludes the null string).

If it is, it substitutes the defined value.

If it is not, it assigns the *default_value* to *parameter* and substitutes its value.

Only shell variables can be assigned values in this way, so $*parameter* cannot be a positional parameter or a special shell parameter.

${*parameter*?[*default_value*]}

The shell checks that *parameter* is defined, i.e. already assigned a value (note that the assigned value may be the null string).

If it is, it substitutes the defined value.

If the condition is not fulfilled and a *default_value* has been specified, the shell aborts execution of the command or shell script and writes the following error message on standard error output: □
parameter: *default_value*

If the condition is not fulfilled and no *default_value* has been specified, the shell aborts execution of the command or shell script and writes the following error message on standard error output:

parameter: `parameter null or not set`

No value is assigned to the parameter in this case.

${*parameter*:?*default_value*}

The shell checks that *parameter* is defined, i.e. already assigned a value (but not the null string; the colon precludes the null string).

If it is, it substitutes the defined value.

If the condition is not fulfilled and a *default_value* has been specified, the shell aborts execution of the command or shell script and writes the following error message on standard error output:

parameter: *default_value*

If the condition is not fulfilled and no *default_value* has been specified, the shell aborts execution of the command or shell script and writes the following error message on standard error output:

parameter: `parameter null or not set`

No value is assigned to the parameter in this case.

${*parameter*+*default_value*}

The shell checks that *parameter* is defined, i.e. already assigned a value (note that the assigned value may be the null string).

If it is, it substitutes the *default_value*.

If it is not, it substitutes the null string.

${*parameter*:+*default_value*}

The shell checks that *parameter* is defined, i.e. already assigned a value (but not the null string; the colon precludes the null string).

If it is, it substitutes the *default_value*.

If it is not, it substitutes the null string.

Example:

```
$ echo ${dir:-`pwd`}
/usr1/carl/src
```

The shell executes `pwd` only if the variable `dir` is undefined or has a value equal to the null string.

COMMAND SUBSTITUTION

In processing the command line the shell replaces a command enclosed in backquotes `...` with the output produced by the command:

`*command*`

command

Name of the command which the shell is to execute; the shell takes the output of this command and substitutes it for `*command*` on the command line.

If the command line consists of `*command*` on its own, the shell attempts to execute the output of *command*.

Processing proceeds in the following sequence:

- the current shell detects the backquotes on the command line and calls a subshell.
- the subshell processes *command* (see **COMMAND LINE PROCESSING** below) and terminates.
- `*command*` is replaced on the command line by the output of *command*.
- the shell parses the thus modified command line again, treating as argument separators the characters assigned to the `IFS` variable.
- the current shell executes the command line.

If you have redirected standard output within the backquotes, `*command*` will be replaced by the null string.

Several commands may be entered within backquotes, separated from each other by command separators.

Example:

You want the shell prompt to tell you the name of the current host:

```
$ PS1="<`hostname`> "
<amadeus>
```

As a result of the double quotes, the blank after > forms part of `$PS1`.

COMMAND INPUT AND OUTPUT REDIRECTION

By default, the standard input of a command is the keyboard, while the standard output and standard error are both assigned to the terminal screen. These three standard I/O files can also be referenced via the following file descriptor values:

`0` for standard input

`1` for standard output

`2` for standard error.

When the shell initiates a particular command in the background, it redirects standard input for this command to the null file `/dev/null` (the bit bucket). However, you can redirect standard input of a background command to a different file.

The default assignments for a command may be altered by specifying the redirection anywhere in the command. The following section explains the various forms of redirection for:

- standard input
- standard output
- standard error.

Redirecting standard input

The less-than sign < redirects standard input. The argument <... does the same as 0<...

<*file* Redirects standard input of the command to *file*. The command reads its input from *file*.

file Name of the file to which standard input is to be redirected. This file must already exist and must be readable by you.

Please note that since the shell has not yet interpreted the metacharacters *, ?, and [...] at this stage, these characters should not be used when specifying *file*.

<&– Closes standard input for the command. This means that the command only receives EOF as its input.

Example:

```
$ wc –l <&–
0
```

<&*digit*

Redirects standard input of the command to the file associated with file descriptor *digit*. The command then reads its input from this file.

digit File descriptor of the file to which standard input is to be redirected.

The character & following < instructs the shell that the next argument is a file descriptor and not a file name.

<<[–]*string*

Introduces a *Here Document*. Shell scripts use Here Documents for commands that read from standard input.

The standard input of the command is redirected to the Here Document, which comprises all the lines in the script between the introductory line and the first line that contains nothing but *string*. If the latter does not exist, the command reads to the end of the script. The command reads its input from the Here Document.

The shell processes the lines of the Here Document, replacing variables with their value and commands in backquotes with their output. The lines of the Here Document are passed to the command after these substitutions have been made (see **COMMAND LINE PROCESSING**, Step 5).

Characters such as \, $, and `...` must be quoted in the Here Document (with a preceding backslash) if they are to be treated literally. Quoting any character in *string*, for example by escaping it with a backslash, causes the entire document to be treated literally, i.e. escapes all special characters in it.

– Causes the shell to strip all leading tabs from *string* and from the Here Document.

– not specified: □
Initial tabs are retained.

string

Any string; must not appear as part of the Here Document itself. The document ends at the first line that contains only the specified *string*. If there is no such line, the end of the script containing the document acts as the end of the document.

Quoting any character in *string*, for example by escaping it with a backslash, causes all shell metacharacters in the Here Document to be escaped automatically.

The occurrence of *string* which marks the end of the Here Document has to be written without the escape symbol.

Example of a Here Document:

The contents of the shell script `post` are as follows (without the line numbers):

```
1 mail $* << STOP
2 To all system administrations!!
3
4 The next meeting on data security is scheduled for
5 Friday, December 13, at 10 a.m. in the conference room
6 on the 6th floor. Please make every effort to attend.
7
8 STOP
```

The Here Document comprises lines 2 to 7; the string `STOP` terminates it. The standard input of the `mail(1)` command is redirected to the document.

Redirecting standard output

The greater-than sign > redirects standard output. The argument >... does the same as 1>...

`>`*file* Redirects standard output of the command to *file*. The command writes its input to *file*.

file Name of the file to which standard output is to be redirected.

If the file already exists, it previous contents are deleted. If the file does not yet exist, it is created.

Please note that since the shell has not yet interpreted the metacharacters *, ?, and [...] at this stage, these characters should not be used when specifying *file*.

`>&-` Closes standard output for the command. The command does not write any output.

Example:

```
$ ls >&-
$
```

`>&`*digit*

Redirects standard output of the command to the file associated with file descriptor *digit*. The command then writes its input to this file.

digit File descriptor of the file to which standard output is to be redirected.

The character & following > instructs the shell that the next argument is a file descriptor and not a file name.

`>>`*file*

Redirects standard output of the command to *file*. The command writes its output to *file*.

file Name of the file to which standard output is to be redirected.

If the file already exists, the output from the command is appended to its previous contents. Otherwise, a new file is created.

Please note that since the shell has not yet interpreted the metacharacters *, ?, and [...] at this stage, these characters should not be used when specifying *file*.

Redirecting standard error

The argument 2> redirects standard error output. The file descriptor 2 must be included.

`2>`*file*

Redirects standard error output of the command to *file*. The command writes all its error messages to *file*.

file See the description under >*file*.

`2>&1` Redirects standard error to the file associated with file descriptor 1 (see description under `>&`*digit*). The error messages and the output from the command are sequentially written to the corresponding file.

`2>>`*file*

Redirects standard error of the command to *file*. The command writes all its error messages to *file*.

file See the description under >>*file*.

Multiple redirections

A redirection can appear at any point on the command line. You can also include several redirections on the same command line:

- You can redirect both standard input and standard output for a command. The two redirections should be to different files, as otherwise the shell would delete the file before the command could read its input.

  Example:

  ```
  $ <list sort >list
  ```

  After execution of this command, the file `list` will be empty. The shell redirects standard input and standard output of `sort` before running `sort` itself. Because of the `>list` argument, the previous contents of file `list` are deleted, so `sort` has no data to sort and has nothing to output.

- You can redirect both standard output and standard error for a command, but in this case the order in which the redirections are listed is significant. The shell evaluates redirections from left to right.

  Example:

  ```
  1>outfile 2>&1
  ```

  The shell first redirects standard output to the file `outfile` and then redirects standard error to the file associated with file descriptor 1 (also `outfile`). This means that the output and error messages of the command in question will both be sent to `outfile`.

  In a different order:

  ```
  2>&1 1>outfile
  ```

  Here the shell first redirects standard error to the file associated with file descriptor 1 (by default, the screen, i.e. the file `/dev/tty`) and then redirects standard output to the file `outfile`. In other words, only the output of the command in question will be written to `outfile`, not the error messages.

FILE NAME GENERATION

In processing the command line, the shell searches for arguments that contain one of the following metacharacters [see **COMMAND LINE PROCESSING**, Step 9 and `expressions(5)`]:

```
* ? [
```

The shell takes these arguments as patterns and matches them against the file names in the appropriate directory. If the shell finds a file name that matches an argument, this file name is substituted for the argument. An argument matched by several file names is replaced by a sorted list of all matching file names. The collating sequence is governed by the value of the `LC_COLLATE` and `LANG` environment variables. Each file name acts as a separate argument.

If no matching file name is found, the shell uses the unaltered pattern as an actual file name.

If the pattern does not contain a slash, the shell always searches for matching file names in the current directory. If the pattern is to match file names that begin with a dot (.), this dot must be specified as the first character of the pattern. By contrast, periods within a file name need not be specified in the pattern.

A pattern for file names is any string including one or more of the following metacharacters:

`*` as a pattern on its own:

The shell replaces the asterisk (*) with the names of all files in the current directory that do not begin with a dot.

If the current directory is empty, the shell uses the `*` character as an actual file name.

as part of a pattern:

The shell replaces the asterisk with zero, one, or more characters to match file names in the current directory. A dot may also be among the characters used to replace the asterisk, but only if the `*` is not the first character in the pattern.

If the current directory is empty, or no file name matches the specified pattern, the unaltered pattern including the * character is treated as an actual file name.

The metacharacter * is not matched by:

- aslash (/) or
- adot (.) when * is the first character in the pattern.

? The shell replaces the ? character with any single character to match file names in the current directory. A ? can also be matched by a dot (.), but only if the ? is not the first character in the pattern.

If the current directory is empty or no matching file name is found in it, the unaltered pattern including the ? character is treated as an actual file name.

The metacharacter ? is not matched by:

- aslash (/) or
- adot (.) when * is the first character in the pattern.

[*s*] Replaced by any one of the characters included in the set *s* to match file names in the current directory. The square brackets are mandatory.

If the current working directory is empty or does not contain any matching names, the shell uses the pattern [*s*] as a file name as it stands.

s *s* is a set of characters which may include any ordinary characters other than a slash (/). If the pattern is intended to match file names beginning with a dot (.), the dot must not be within the square brackets. Please note that an exclamation point (!) as the first character and a hyphen (–) anywhere within square brackets have a special meaning. The * and ? characters do not serve as wildcard characters when enclosed within square brackets.

[!*s*] Replaced by any single character other than those in set *s* to match file names in the current working directory. The square brackets are mandatory.

If the current working directory is empty or does not contain any matching names, the shell uses the pattern [!*s*] as a file name as it stands.

s as for [*s*].

[*c1–c2*]

Replaced by any one character from the bracketed range to match file names in the current directory. The square brackets are mandatory.

If the current working directory is empty or does not contain any matching names, the shell uses the pattern with its brackets as a file name as it stands.

c1–c2 specifies a range of characters which includes all characters that lie between *c1* and *c2* in the ASCII character set. *c1* and *c2* are included in the range. You can also list a number of ranges, with no separators between them.

c1 *c1* can be any ordinary character other than a slash (/).

If the pattern is to match file names that begin with a dot, the dot must be the first character of the pattern, i.e. it must come before the left square bracket.

Please note that an exclamation mark (!) as the first character and a dash (–) anywhere within square brackets have a special meaning.

c2 *c2* can be any ordinary character other than a slash (/).

[!*c1–c2*]

Replaced by any single character other than those in the bracketed range to match file names in the current directory. The square brackets are mandatory.

If the current working directory is empty or does not contain any matching names, the shell uses the pattern with its brackets as a file name as it stands.

c1–c2 specifies a range of characters (see description of [*c1–c2*].

[:*class*:]
Character class expression. Any character of type *class*.

Character class expressions must always appear within a second pair of square brakkets, e.g. [*c*[:digit:]].

The character classes in an internationalized environment are defined by the environment variables LC_TYPE and LANG.

class is one of the following:

alpha any letter

upper any uppercase letter

lower any lowercase letter

digit any decimal digit (0 to 9)

xdigit any hexadecimal (0 to 9, a to f and A to F)

alnum any alphanumeric character (letter or digit)

space any character producing white space in displayed text (whitespace characters = blanks and tabs)

punct any punctuation character

print any printable character (including the characters in space)

graph any character with a visible representation (excluding the characters in space)

cntrl any control character

[=*c*=] Equivalence class expression. Any character or collating element defined as having the same relative order in the current collating sequence as *c*.

Equivalence class expressions must always appear within a second pair of square brackets, e.g. [[=a=]bc].

The collating sequence in an internationalized environment is defined by the environment variables LC_COLLATE and LANG.

[.*cc*.]
Collating symbol. Multi-character collating elements must be represented in this form to distinguish them from ordinary characters. An expression of this type is collated as a single character. *cc* has to be defined as a valid collating element in the current collating sequence.

Collating symbols must always appear within a second pair of square brackets, e.g. [c[.ch.]d].

The collating sequence in an internationalized environment is defined by the environment variables LC_COLLATE and LANG.

Within the square brackets you can, without separators, concatenate any set of *c1–c2* ranges, single characters, collating symbols [.*cc*.], character class expressions [:*class*:] and equivalence class expressions [.*cc*.]. An exclamation point ! used to exclude specific characters or ranges must be the first character after the left square bracket [. An exclamation point appearing anywhere else within the brackets has no special meaning.

Caution:
The current character set and the current collating sequence together determine which characters form part of a [*c1–c2*] range. For example, in the locale LANG=LC_COLLATE=En_US.ASCII the command rm [a-z]* deletes all files with names beginning with a lowercase letter from a to z inclusive. In a number of other locales the range a-z also include uppercase letters, and in these environments the same command would also delete files with names beginning with a capital letter. What is more, there are languages, Norwegian, for example, where some letters collate after z. So in an internationalized environment you should always represent character classes such as lowercase letters using

expressions of the form `[:class:]`. Thus instead of the command given above you would use `rm [[:lower:]]*`.

Shell scripts written in a non-internationalized environment (or without allowing for the problems described above) should only be run in a non-internationalized environment; in other words, the NLS variables `LANG`, `LC_COLLATE` and so on should be undefined or null.

COMMAND LINE PROCESSING

In an interactive shell you terminate a command line and pass it to the shell by hitting CR. The portions in which the shell handles input are governed by the characteristics of your terminal [see `stty(1)`]. The input to a script shell consists of the contents of the script being read.

The command line may be made up of a number of commands separated by command separators. In a shell script, newline characters are also permitted as command separators. The shell processes each individual command in succession and executes it. Hence in the following description the term "command line" refers to the logical unit bounded by unescaped command separators. The shell executes all the commands in the input before issuing its prompt again.

The shell processes a command line in several distinct steps, each of which is performed once per command line. The sequence in which these steps are performed is fixed. This order must be taken into account so as to ensure that the shell processes your inputs as you require.

The way in which a command line is processed also depends on the options that you set when calling this shell and those defined later with the `set` command.

Here are the individual steps in chronological order:

1. The shell reads the command line up to the first unescaped command separator and looks for backslashes, single and double quotes and backquotes.
2. It checks the command line for I/O redirections and variable assignments.
3. It breaks the command line down into individual arguments (parsing).
4. It substitutes values for specified variables.
5. It replaces commands between backquotes with the output they produce.
6. It redirects standard input and/or standard output and/or standard error to the specified files.
7. It makes assignments to variables.
8. It takes the line thus modified and parses it again, on the basis of the argument separators assigned to the `IFS` variable.
9. It performs file name generation on arguments including the following characters or strings: * ? [...] [!...]
10. It then executes the command.

Steps 5, 6, and 7 are not executed unless the relevant metacharacters are detected by the shell in Steps 1 and 2.

Step 1: Reading up to the first command separator

The shell reads in the command line (from standard input or from a file) up to the first command separator, which may be any of the following:

`newline ; & | && ||`

If you press the CTRL-D key instead of entering a command when the shell issues its prompt, the current shell exits. A subshell running a shell script exits after executing the last command in the script, i.e. on reading the end-of-file character (EOF). This does not apply when you run a shell script with the Bourne shell built-in . (dot), however; the current shell running the script in this case does not exit when it reads EOF.

The shell then searches its input for:

- escape symbols:

  `\ '...'"..."`

  Metacharacters escaped by any of these quoting mechanisms are henceforth protected from interpretation

by the shell (see **Quoting shell metacharacters** above).

- backquotes:

  `` `command` ``

  In Step 5 this expression is to be replaced by the output of command (see **COMMAND SUBSTITUTION** above). Step 5 (command substitution) is omitted if the shell does not find any backquotes in Step 1.

  The shell protects *command* against interpretation by the shell, as if the whole expression were within double quotes: "*command*".

If the –v option has been set in the current shell, this shell will output the command line after the first step [see set(1)].

Step 2: Locating the metacharacters > < and =

The shell searches for the following metacharacters, which initiate specific actions in subsequent processing steps:

> or <
: redirects standard input, standard output, or standard error in Step 6 in accordance with the specifications that follow (see **COMMAND INPUT AND OUTPUT REDIRECTION** above).

=
: assigns a value to a variable in Step 7 (see **SHELL VARIABLES** above).

The above actions are omitted if the shell fails to find the corresponding metacharacters at this point.

Step 3: Parsing into arguments

The shell now breaks the command line down into arguments using the following argument separators:

blanks tabs

It then strips all argument separators from the command line.

The shell does not access the value of the IFS variable until the eighth step, when it parses the command line again.

The shell also checks whether the first command line argument (the argument immediately following a command separator) is one of the following keywords or metacharacters (see **THE SHELL AS PROGRAMMING LANGUAGE** below):

```
if elif case for while until { (
```

If so, it strips the keyword/left parenthesis from the command line and then repeats Steps 1 to 3 for subsequent command lines until it encounters a command line beginning with the partner keyword or right parenthesis:

```
fi esac done } )
```

If a syntax error is detected, an error message is issued and further processing is aborted. □

Step 4: Substituting values for variables

The shell now searches for the $ character followed by an admissible variable name:

`$0 $1 $2 ... $9 $* $@ $# $? $! $$ $- $`*name* `${...}`

Values are then substituted for the corresponding variables, with the null string being substituted for undefined variables (also refer to **SHELL VARIABLES** and **SHELL PARAMETERS** above).

Step 5: Command substitution

The shell processes strings between backquotes `...`. However, this only applies to backquotes detected by the shell in Step 1:

`` `command` ``

If you have redirected standard output within the backquotes, `` `command` `` will be replaced by the null string.

Several commands may be entered within backquotes, separated by command separators (also refer to **COMMAND SUBSTITUTION** above).

Step 6: Redirecting standard input/standard output/standard error

The shell now processes command line arguments in which it found one of the following characters in Step 2:

`< <&- <& << <<- > >&- >& >>`

Having performed the redirection, the shell strips the redirection arguments from the command line. The command itself remains unaware of the redirection (also refer to **COMMAND INPUT AND OUTPUT REDIRECTION** above).

Step 7: Assigning values to variables

The shell processes those points in the command line at which it found an equals sign (=) in Step 2 (also refer to **SHELL VARIABLES** and **THE ENVIRONMENT** above).

The command line processed to this point may now look something like this:

name=value ...

If the command line contains only an assignment and no command name, the current shell creates the *name* variable and assigns *value* to it. There may be more than one assignment on a command line.

Once the shell has assigned values to all the given variables, no more processing is done on the command line. In this case Steps 8 to 10 are omitted.

From this point on, the newly defined variables will be recognized in the current environment. If they are also to be passed to a subshell, you will need to export them [see the Bourne shell built-in `export(1)`]. Variables are automatically exported if the `–a` option is set [see the Bourne shell built-in `set(1)` and the `sh` command options].

name=value ... command

If the command line contains a command name in addition to an assignment, the current shell does not add the specified variable to the current environment, but only to the environment for *command*. The command name may be preceded by more than one assignment.

The shell then strips the assignments from the command line and runs *command* in the new environment. □

If you have used the `set` command to set the `–k` option in the current shell, assignments may also follow the command name [see `set(1)`].

Step 8: Second parse run

The command line is now broken down into arguments again, on the basis of the argument separators assigned to the `IFS` variable. The default input field separators are:

blank tabs newline characters

Newline characters are no longer interpreted as command separators in Step 8.

A command line needs to be parsed again due to the substantial changes that it has undergone since Step 3:

- variables have been replaced by their values (Step 4).
- commands in backquotes have been replaced by their output (Step 5).
- keywords and parentheses have been stripped from the command line (Step 3).
- redirection and assignment arguments have been stripped from the command line (Steps 6 and 7).

The shell then strips all argument separators from the command line, as well as any null strings that are not enclosed in single or double quotes. Such unprotected null strings are produced when the shell replaces an undefined variable with the null string. Null strings specified in the form "" or ''are retained as arguments.

Following this parse run, the first argument consists of the command name, while the remaining arguments constitute actual arguments to the command.

Step 9: File name generation

The shell looks for arguments that contain any of the following metacharacters:

`* ? [`

The shell takes these arguments as patterns and matches them against the file names in the appropriate directory (see **FILE NAME GENERATION** above).

If the shell finds a file name that matches an argument, this file name is substituted for the argument. An argument matched by several file names is replaced by a sorted list of all matching file names. Each file name

acts as a separate argument.

If no matching file name is found, the shell uses the unaltered pattern as an actual file name.

If the `-f` option has been set in the current shell, this processing step is omitted. If the `-n` option has been set in the current shell, the shell will print the command line after Step 9, omitting Step 10 [see `set(1)`].

Step 10: Executing the command

The command line now contains only arguments that are required for the execution of the command in question. The shell interprets the first argument as the name of the command to be executed. The remaining arguments are interpreted as arguments to this command and are passed to the command on invocation.

The shell tests the command name in the following order:

- Is the command a Bourne shell built-in?

  If it is, the shell itself executes the command.

  If not, it proceeds to the next test.

- Is the command a shell function?

  If it is, the shell itself executes the function.

  If not, the shell assumes that it is dealing with an external command. It therefore runs `fork(2)` to create (spawn) a new process. This process is a copy of the calling shell and thus inherits its environment. This new shell process now performs further tests:

- Does the command name include a slash character / ?

  If it does, the shell process attempts to run the command under the given name using `exec(2)` and without accessing the `PATH` variable.

  The command with a name including a slash will be rejected immediately by a restricted shell (see `sh`, option `-r`).

  If there is no slash in the command name, the shell looks for the name in the current hash table [see `hash(1)`]:

  - if it is there, the shell process attempts to run the command under the absolute path name given in the hash table, using `exec(2)` and without accessing the `PATH` variable.
  - if the command name is not in the hash table, the shell process reads the `PATH` variable and forms an absolute path name from the first path name listed there, a slash and the command name. It then attempts to run the command under this absolute path name using `exec(2)`.

- Can the `exec(2)` system call load the file?

  If it can, the command is executed. With commands that are to run in the background, the calling shell immediately returns and issues its prompt. With all other commands, it waits for them to finish executing.

  With commands called under their basename (without path prefix), the hash table is then updated. A command which is not yet in the table is entered in the following format:

  □

If the command was already in the hash table, the value in the `hits` column on the corresponding line is simply incremented by 1 [see `hash(1)`].

If `exec(2)` cannot load the file, it will be for one of the following reasons:

– The named file is not a binary file (e.g. in `a.out` format). In this case the file must be a shell script. The current shell process reads the file and executes the commands in the steps described here.

– The user invoking the command does not have execute permission for the file. In this case the calling shell returns with the error message:

command`: execute permission denied.`

– `exec(2)` cannot find the file. If the name used to invoke the command contained a slash, or if the command was already in the hash table (so the `PATH` variable was not consulted), the calling shell returns with the error message:

command: not found.

Otherwise, the shell process again reads the PATH variable. This time it attempts to exec(2) the command under an absolute path name formed using the second path name listed in the PATH variable.

If exec(2) still cannot find the file, the procedure is repeated using the next path name listed in the PATH variable, and so on until all the listed path names have been tried.

If exec(2) still has not found the file after all the path names listed in the PATH variable have been tried, the calling shell returns with the error message:

command: not found.

THE SHELL AS PROGRAMMING LANGUAGE

The shell provides the essential elements of a programming language:

- variables
- parameters
- built-in functions (= Bourne shell built-ins)
- comments
- macros (= shell functions)
- control-flow statements.

Every command line that you enter at the shell prompt is a small shell program in itself. The shell interprets this line on the basis of specific rules and executes the corresponding command (see **COMMAND LINE PROCESSING** above).

To write more extensive programs you use an editor. This entails creating a file and entering in it all the commands that are to be sequentially executed by the shell. Programs like this, known as shell scripts or shell procedures, do not need to be compiled. When you invoke a shell script with sh *scriptname*, the shell starts a subshell, which reads, interprets, and executes each line in the file. Once all the commands in this file have been executed, the subshell exits and the parent shell returns.

This section provides details on:

- comments in shell scripts,
- defining and invoking shell functions,
- shell control-flow statements (in alphabetical order).

Comments in shell scripts

Comments are introduced by a hash character (#). All characters that follow the # up to the first newline (escaped or unescaped) are ignored by the shell.

Example:

Ways of entering and escaping comments in shell scripts:

```
date # comment□
echo \# does not begin a comment□
pwd # comment□
date # this comment ends here -> \□
echo \# a comment cannot extend over more than one line
```

If the # is escaped, its special meaning as a comment character is lost.

The Bourne shell built-in : (colon) provides you with an additional method of introducing a comment. This command returns an exit status of 0 and does nothing (null command).

The comment that follows the colon must be separated from it by a blank or a tab. The following must be observed when you use : to introduce comments:

- The shell will normally parse elements of the comment as command line arguments, among other things substituting values for any shell parameters that it encounters. Enclosing a comment in single quotes

escapes any shell metacharacters in it.

- The colon must always be the first character in a line or follow a command separator such as a semicolon. Otherwise, it will not be recognized as a command by the shell.
- ```
  : This shell script displays comments \
  and prints the PID of the shell and the date
  echo $$; : '$$ is the PID of the current shell'; date
  ```
  With `:` you can have comments extending over a number of lines and insert them between two commands on the same line.

SHELL FUNCTIONS

A shell function combines several commands under a single name.

This section describes:

- how to define a shell function,
- how to execute a shell function,
- the differences between shell functions and shell scripts, and
- the Bourne shell built-in `return`.

Defining a shell function

The definition of a shell function has the following format:

name `()` { *commandlist* `;` }

name Name of the shell function. A function name may consist of letters, digits, and underscores in any order, but must not begin with a digit. *Note*: Names of built-in shell commands must not be used in accordance with the X/Open specification [cf. list in section **BOURNE SHELL BUILT-INS** above and under `ksh(1)`].

You cannot use the same name for a shell function and a shell variable. If you define a shell function with a name already given to a shell variable, the value of the shell variable will be lost, and vice versa.

commandlist

One or more commands such as are allowed in shell scripts. Commands are separated by newline characters or other command separators. A shell function can, in turn, invoke or define further shell functions.

The commands in *commandlist* are executed when you call the shell function.

The blank after the left brace and the semicolon before the right brace may both be replaced by newlines.

Example:

Defining the shell function `ll` interactively:

```
$ ll() {
> ls -l $*
> }
```

The shell function `ll` can now be used to simulate the `ls -l` command.

Once a shell function has been defined, it is known in the current shell. The Bourne shell built-in `set(1)` writes all the shell variables (with their values) and functions defined in the current shell on standard output. The definition of a shell function can also be displayed by calling the Bourne shell built-in `type(1)` with the function name as an argument.

One limitation of shell functions is that they cannot be exported and are hence only recognized by the shell in which they were defined. Shell function definitions can be removed with the `unset` command.

All variables known in the current shell can be accessed from within a shell function. As in the case of shell scripts, positional parameters may also be passed with the function call. However, the positional parameter $0 does not contain the function name but the invocation name of the current shell or the name of the shell script, depending on where you called the function.
```

Defining shell functions in a file

If you plan to use a shell function frequently, it is advisable to write its definition into a file. This file can then be executed in the current shell using the Bourne shell built-in . (dot). The function defined in this file will then be accessible in the current shell. Multiple function definitions can also be collected together in one file.

Executing a shell function

You invoke a shell function in the same way as a command. A shell function is executed by the current shell itself, i.e. no subshell is created as for shell scripts.

Example:

```
$ ll /dev
```

The shell function `ll` which we defined above is invoked. The shell sets shell parameter `$*` to the directory name `/dev`.

As with a shell script, you can pass arguments to your shell function. The current shell sets the positional parameters to these arguments. In other words, by calling a shell function you redefine the positional parameters in the current shell (as with `set`).

The only limitation is that a shell function can only be called from the shell in which it was defined.

Variable assignments within a shell function change the environment of the current shell.

Differences between shell functions and shell scripts

Shell functions are more efficient than shell scripts. Like Bourne shell built-ins, they are given preferential treatment by the shell, as it does not have to create a new subshell and is not required to load data from a file first.

A shell function is bound to the shell in which it was defined.

Every invocation of a shell function redefines all positional parameters.

The Bourne shell built-in `return(1)` can be used to make shell functions exit with a predefined exit status. This command can only be invoked from within a shell function. By default, a shell function exits when all the commands in it have been executed.

Exit status of a shell function

By default, the exit status of a shell function is the exit status of the last command executed within the function. This exit status can be checked with `echo $?`.

If you want a shell function to exit with a predefined exit status, you should use the Bourne shell built-in `return(1)` . As mentioned earlier, this command can only be invoked from within a shell function.

As in shell scripts, you could also use the Bourne built-in `exit(1)` to define an exit status. However, the `exit` command terminates not only the shell script/shell function but also the executing shell.

Exiting a shell function

The Bourne shell built-in `return` is used to exit from shell functions. It can be invoked with a number to define an exit status for the corresponding function.

Shell functions also exit without `return`:

- when all the commands have been executed. The exit status is then that of the last command executed.
- when the executing shell cannot find a command or detects a syntax error; the exit status is then non-zero.

`return` can only be called from within a shell function.

`return` [*exit_status*]

exit_status

Any number between 0 and 256. This is the exit status returned by the shell function.

If you specify a larger number, the exit status will not be the specified number but the integer remainder of dividing the specified number by 256.

The exit status can be checked with `echo $?`.

exit_status not specified:

The exit status of the shell function is that of the command last executed before `return` was invoked.

Example:

The `false_new` command can be implemented by means of the following shell function:

```
false_new() {
return 255
}
```

Shell control-flow statements

Control-flow statements form a part of every programming language. The following statements are provided by the shell:

`break` loop control

`case ... esac`
test and branch

`continue`
loop control

`for ... do ... done`
loop through list

`if ... then ... elif ... then ... else ... fi`
test condition and execute commands

`until ... do ... done`
loop with exit condition

`while ... do ... done`
loop with exit condition

These control-flow statements can either be entered interactively or used in a shell script. The following keywords are only recognized by the shell if they are entered at the beginning of a command (i.e. after a command separator) and if they are not escaped:

```
case  for
do    if
done  then
elif  until
else  while
esac  {
fi    }
```

The braces serve to collect the output of commands enclosed within them (see **Command grouping** above).

The control statements `if`, `until`, and `while` first test for a condition. In the shell, conditions can only be given in the form of a command or a command list (see **Command lists** above). The shell then evaluates the exit status.

- A condition is held to be true if the command or command list that defines it returns an exit status of 0.
- A condition is held to be false if the command or command list that defines it returns a non-zero exit status.

Control-flow statements can be freely combined and nested.

break – exit from loop

The Bourne shell built-in `break(1)` exits from the enclosing `for`, `until`, or `while` loop. In a shell script, control passes to the command following the `done` at the end of the broken loop.

Loops may be nested to any level. If you wish to use `break` to exit from several nested loops, you can do so by calling `break` with the appropriate number.

`break` [*n*]

n The number of nested `for`, `until`, or `while` loops to be broken by `break`.

n not specified:

`break` terminates the immediately enclosing `for`, `until`, or `while` loop; in other words, *n* defaults to 1.

case – test and branch

The `case` statement can be used to compare input with specified patterns and branch to commands on the basis of the result.

```
case name in
)
pattern[|pattern ...]) [commandlist];;
.
.
.
pattern[|pattern ...]) [commandlist]
esac
```

name Any arbitrary string, typically a shell parameter. The shell compares *name* with the specified pattern.

pattern

String to be compared with *name*. The form of the patterns is the same as that used for file name generation [see **FILE NAME GENERATION** above and `expressions(5)`] except for the restrictions on dots and slashes which apply there.

Metacharacter	**Matches**
*	any string of zero, one or more characters, including dot and slash
?	any one character, including dot and slash
[*s*]	any character included in set *s* (after evaluation of [.*cc*], [=*c*=] and [:*class*:]). *s* may include: – any ordinary characters (including dot . and slash /) – collating symbols [.*cc*.] – character class expressions [:*class*:] – equivalence class expressions [=*c*=]
[!*s*]	any character other than those included in set *s* (after evaluation of `[.cc]`, `[=c=]` and `[:class:]`).
[!*c1–c2*]	any one character from bracketed range; *c1* and *c2* are part of the range. *c1* and *c2* can be: – any ordinary characters (including dot . and slash /) – collating symbols [.*cc*.] – equivalence class expressions [=*c*=]
[*c1–c2*]	any one character other than those in the bracketed range.

If *name* matches the given pattern, the shell executes the command list specified after it. The `case` statement ends after executing these commands.

If *name* does not match this pattern, the next pattern is checked.

If the same command list is to be executed for multiple patterns, all such patterns can be entered before the command list, separated from one another by a | (vertical bar).

If you specify the character * for *pattern*, any patterns that come below it will not be checked; `case` will

always execute the command list after the * pattern and exit.

commandlist
List of commands to be executed if *name* matches the corresponding pattern.

Every command list other than the last must be terminated with two semicolons.

If you do not want any commands to be executed, you must at least include the two semicolons, except in the case of the final pattern.

A `case` statement is terminated with the keyword `esac`.

continue – exit from loop

The control-flow statement `continue` terminates the current iteration of the enclosing `for`, `until`, or `while` loop. The loop resumes at the next iteration. A shell script likewise resumes at the next iteration of the same loop.

Loops may be nested to any level. If you wish to resume at the *n*-th enclosing loop, you can do so by invoking `continue` with the appropriate number.

`continue` [*n*]

n Number of the enclosing `for`, `until`, or `while` loop at which the shell script is to resume.

n not specified: □
The `continue` statement resumes the shell script at the next iteration of the enclosing `for`, `until`, or `while` loop.

for – loop through list

The `for` statement allows you to have commands in a list repeated several times. The loop is traversed once for each argument in the specified list.

`for` *name* [`in` *list*] □
`do` *commandlist* □
`done`

name Name of a shell variable to which the shell successively assigns the value of each argument in the specified *list*. The given *commandlist* is executed after each assignment. The keyword parameter `$`*name* references the value currently assigned to the *name* variable.

list List of arguments; the argument separators are the characters assigned to the `IFS` variable. Each argument is successively assigned as a value to the *name* variable, and the *commandlist* is executed once for each argument. When the *list* of arguments is exhausted, the `for` statement terminates.

If an argument contains any of the metacharacters *, ? and [...], it is replaced by matching file names (see **FILE NAME GENERATION** above).

list not specified:

`in` is also omitted. The shell parameter `$@` is assumed as the default, so the `for` loop is executed once for each argument used in calling the shell script or defined by `set` in the current shell.

commandlist
List of commands to be executed as often as required.

A `for` loop terminates at the keyword `done`.

if – test condition and execute commands

The `if` statement allows you to test conditions and execute specific commands, depending on the result.

The shell first executes the command list that is specified directly after `if`:

- the condition is then true if the exit status returned by the final command in the list is 0.
- the condition is false if a non-zero exit status is returned by the final command.

`if` *commandlist1* □
`then` *commandlist2* [`elif` *commandlist3* □
`then` *commandlist4* ...] □
[`else` *commandlist5*] □

`fi`

- Sequence of commands to be executed by the shell in order to test the condition.

  If the condition is true, the shell branches to `then`; if false, to the next `elif` or `else`.
- Sequence of commands to be executed by the shell if the condition following `if` is true. The `if` statement ends after the execution of these commands.
- Sequence of commands to be executed by the shell to test the next condition. The shell only executes *commandlist3* if the condition following `if` is false.

  If this condition is true, the shell branches to the next `then`; if false, to the next `elif` or `else`.

  The `elif` branch is optional. If it is omitted, the `then` following it is omitted as well.
- Sequence of commands to be executed by the shell if the condition following `if` is false and the condition following `elif` is true. The `if` statement is terminated after completion of these commands.
- Sequence of commands to be executed by the shell if the conditions following `if` and `elif` were false. The `if` statement terminates after execution of these commands.

  If the `else` segment is omitted, the shell does not execute any commands unless one condition is true.

An `if` statement is terminated by the keyword `fi`.

The `if` statement returns an exit status of 0 if the shell has not executed commands from a `then` or `else` branch, i.e. when no condition is satisfied.

until – loop with exit condition

In the shell there are two loops with which an exit condition can be given: `until` and `while`.

The `until` statement checks whether a specified condition is false. The loop terminates as soon as the condition becomes true.

`until` *commandlist1* □
`do` *commandlist2* □
`done`

commandlist1

Sequence of commands executed by the shell in order to test the condition.

If the condition is true, the `until` statement terminates, and the commands from *commandlist2* are not executed.

If the condition is false, the shell executes the commands from *commandlist2*. The shell then tests the condition again by executing the commands from *commandlist1*.

commandlist2

Sequence of commands executed by the shell as long as the condition is false.

An `until` statement is terminated with the keyword `done`.

If the commands after the keyword `do`, i.e. in the body of the loop, are not executed at all, the exit value of the `until` statement is 0. This means that the condition was false the first time it was tested.

while – loop with exit condition

In the shell there are two loops with which a termination condition can be given: `while` and `until`.

The `while` statement checks whether a specified condition is true. The loop terminates as soon as the condition becomes false.

`while` *commandlist1* □
`do` *commandlist2* □
`done`

commandlist1

Sequence of commands executed by the shell in order to test the condition.

If the condition is true, the shell executes the commands from *commandlist2*. The shell then tests the condition again by executing the commands from *commandlist1*.

If the condition is false, the `while` statement terminates, and the commands from *commandlist2* are not

executed.

commandlist2

Sequence of commands executed by the shell as long as the condition is true.

A `while` statement is also terminated with the keyword `done`.

If the commands after the keyword `do`, i.e. in the body of the loop, are not executed at all, the exit value of the `while` statement is 0. This means that the condition was false the first time it was tested.

FURTHER DIFFERENCES BETWEEN KORN SHELL – BOURNE SHELL

Since the Korn shell (`ksh`) and Bourne shell (`sh`) do not always behave in the same way, the shell scripts also behave differently [see also section "Major Korn shell features not available in the Bourne shell" in `ksh(1)`].

The following differences are known to exist at present:

1. If an `exit` is executed in a loop (`for`, `until`, `while`) and if the input is read from a file, the Bourne shell quits the loop and the Korn shell ends the script. Example of a `while` loop:

```
while read i
do
        echo $i
        if [ "$i" = "end" ]
        then
              exit 199
        fi
done < inputfile
retflag=$?
echo $retflag
```

2. The Shell `cd` built-in behaves differently when metacharacters (*, ?) are used.

   The current directory contains the subdirectories `dir_cmd`, `dir_lib` and `dir_head`.

   ```
   cd dir_*
   ```

   The Korn shell generates the error message "bad argument count", and the Bourne shell changes to the directory with the first pattern matching (`dir_cmd`).

3. Call parameters are interpreted differently.

   Invoking at the shell level:

   ```
   ((/tmp/shelltest add "-po XON") 2>&1)
   ```

   Korn shell:

   $1 = `add` $2 = `-po` $1 = `XON`

   Bourne shell:

   $1 = `add` $2 = `-po XON`

SEE ALSO

`cd(1)`, `colon(1)`, `dot(1)`, `echo(1)`, `eval(1)`, `exec(1)`, `exit(1)`, `export(1)`, `hash(1)`, `pwd(1)`, `read(1)`, `readonly(1)`, `set(1)`, `shift(1)`, `test(1)`, `times(1)`, `trap(1)`, `type(1)`, `ulimit(1)`, `umask(1)`, `unset(1)`, `wait(1)`.

`csh(1)`, `jsh(1)`, `ksh(1)`, `rksh(1)`, `xpg4_sh(1)`.

`exec(2)`, `exit(2)`, `signal(2)`, `ulimit(2)`, `umask(2)`, `wait(2)`.

`expressions(5)`, `specialchar(5)`.

Kochan, Wood: *UNIX Shell Programming*.

Manis, Meyer: *The UNIX Shell-Programming Language*.

User's Reference Manual

shift(1)

NAME

`shift` - shift values of positional parameters to the left

SYNOPSIS

`shift [`*n*`]`

DESCRIPTION

The shell built-in `shift` shifts the values of positional parameters over to the left. `shift` without arguments thus results in the following:

- Shell parameter `$0` (the command name) is unaffected.
- The original value of `$1` is lost; this value can no longer be accessed.
- Instead, `$1` takes the original value of `$2`, `$2` that of `$3`, and so on until `$8`, which takes the value of `$9`.
- The tenth command-line argument is passed to the last positional parameter `$9`.
- `$#` is reduced by one.
- `$*` and `$@` contain all command-line arguments, starting with the new value of `$1`. The original value of `$1` is dropped.

By default, the shell provides positional parameters which give you direct access to only the first nine command-line arguments of a `set` command or a shell script. This restriction can be bypassed with `shift`, since it allows you to move values over to the left by the required number of places.

OPERANDS

n A positive integer; `shift` is executed *n* times. This means that positional parameter `$1` takes the value of the (*n*+1)th command line argument, etc.

If the value of *n* is too large, `shift` issues an error message as soon as no command-line argument is available for `$1`; i.e. `$#` is equal to 0.

n not specified:

`shift` is executed once.

If `$1` already contains the last argument, the next invocation of `shift` will result in an error message.

ERROR MESSAGES

`script: cannot shift`

This error message is issued when `$#` is equal to 0, i.e. when no value could be passed to `$1`.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The tenth command-line argument is to be accessed in a shell script:

```
.
.
arg1=$1
shift
```

```
arg10=$9
.
.
.
```

The original value of `$1` is assigned to the `arg1` variable. The value thus remains accessible even after the `shift` command.

Example 2

The following interactive session demonstrates the changes that take place in the values of the shell parameters `$1`, `$*`, and `$#` when `shift` is invoked:

```
$ set 1 2 3 4 5 6 7 8 9 10 11 12
$ echo $1
1
$ echo $#
12
$ echo $*
1 2 3 4 5 6 7 8 9 10 11 12
$ shift 6
$ echo $1
7
$ echo $*
7 8 9 10 11 12
$ echo $#
6
```

NOTES

Some differences in behavior may occur when using `shift`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh`(1), `set`(1), `sh`(1).

User's Reference Manual

shl(1)

NAME

`shl` - shell layer manager

SYNOPSIS

`shl`

DESCRIPTION

`shl` is a shell layer manager that allows you to simultaneously interact with more than one shell from a single terminal.

You can control the behavior of each of these shells, known as layers, using the commands described below.

You are free to decide which type of shell (`sh`, `jsh`, `ksh` or `csh`) to use in each layer.

`shl` requires at least one entry for a virtual terminal in the `/dev/sxt` directory. These entries consist of the digits in the name of a virtual terminal. If no such entry exists, `shl` will exit with an error message.

Functionality

`shl` is a management program with which you can create and control individual layers. It binds each layer to a virtual terminal device (`/dev/sxt`) which can be manipulated with `stty` and `ioctl` like a normal terminal line.

When you invoke `shl`, you are immediately prompted to enter commands to create at least one layer. The `shl` manager uses its own prompt `>>>`, to help to distinguish it from a layer. After creating a layer, the layer manager passes control to the shell of that layer, which displays its `PS1` prompt variable as set by `shl` (see below).

This layer is called the current layer, and the shell called by `shl` is the process group leader as in the case of a `login` shell. Each layer thus has its own process group ID. The type of shell called in the layer is determined by the value of the `SHELL` variable. You can have up to seven layers running concurrently.

The input from your terminal is read either by the layer manager or by the current layer. If another layer process attempts to read from standard input, i.e. from the terminal, it is blocked until its layer is made the current layer. The stty character `swtch` (set to <CTRL-Z> if undefined) can be used to switch control from a current layer to the layer manager at any time.

All layers can write their standard output to the terminal, unless you have explicitly blocked the output of individual layers.

You can do this by entering an appropriate `shl` command or by invoking the `stty` command with the `loblk` (layer output blocking) option. The block can be removed by setting the `stty` option `–loblk` or by activating the layer.

Caution:

If you are using a shell with job control, the `swtch` character must be redefined (see `stty`) to avoid disabling the `suspend` character.

Commands

The following commands may be issued in response to the `>>>` prompt of the shell layer manager. Any unique abbreviation (prefix) may be used. The names `(1)` through `(7)` may be abbreviated to just the digit.

A *name* is a sequence of characters delimited by a blank, tab or newline. Only the first eight characters are significant. The names `(1)` through `(7)` cannot be used when creating a layer (see `create` below). They are used as default names by `shl` when no name argument is supplied to `create`.

`create` [*name*]

Creates a layer called *name* and makes it the current layer.

If no *name* argument is supplied, `shl` assigns a name formed by taking the last digit of the name of the

virtual device bound to the layer and placing it in parentheses, e.g. `(3)` for `/dev/sxt/003`.

The shell prompt variable `PS1` is set to the name of the layer followed by a space. A maximum of seven layers can be created.

`block` *name* [*name* ...]
When the named layer is not the current layer, standard output of its processes will be blocked. This command is equivalent to running `stty loblk` (layer output blocking) within the layer.

`delete` *name* [*name* ...]
Terminates the named layer(s). All processes in the process group of the named layers are sent the `SIGHUP` signal [see `signal(2)`].

`help` or `?`
Writes the syntax of the `shl` commands on standard output.

`layers` [`-l`] [*name* ...]
Lists the name and process group ID of each named layer on standard output.

The `-l` option produces a detailed listing similar to that of the `ps` command. If no *name* arguments are supplied, information for all existing layers is listed on standard output.

`resume` [*name*]
Makes the named layer the current layer. If no *name* argument is supplied, the last existing current layer will be activated.

`toggle`
Activates the layer that was current before the last current layer.

`unblock` *name* [*name* ...]
For each *name*, standard output of processes in the corresponding layer will not be blocked when it is not the current layer. This command is equivalent to running `stty -loblk` (layer output blocking) within the layer.

`quit` This command terminates `shl`. The `SIGHUP` signal is sent to the processes of all layers.

name The layer referenced by *name* is made the current layer.

ENVIRONMENT VARIABLE

`SHELL`
Variable defining the command interpreter to invoke.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Typical sequence of operations performed with `shl`:

```
$ shl
create one
one
...<CTRL-Z>create
(3)
...<CTRL-Z>layers -l
_____
_____
_____
(3)<CTRL-Z>toggle
one
```

```
...<CTRL-Z>quit
$
```

FILES

`/dev/sxt/??[0-7]`
Virtual terminal device.

SEE ALSO

sh(1), stty(1), ioctl(2), signal(2), sxt(7).

User's Reference Manual

sinfilt(1)

NAME

`sinfilt -` make files with language-dependent special charactersreadable under Reliant UNIX

SYNOPSIS

`sinfilt` *–char_set* [–o *outfile*] [*file ...*]

DESCRIPTION

`sinfilt` converts MS-DOS files containing language-dependent special characters (such as ä, ö, ü, ß) or data in the IBM character set so that they can be read and edited under Reliant UNIX [see `ascii(5)` and `meta-ascii(5)`]. By default `sinfilt` writes to standard output.

OPTIONS

char_set

You can select one of the following character sets:

`–a`	Danish
`–b`	Belgian
`–d`	German
`–e`	Spanish
`–f`	French
`–g`	British
`–i`	Italian
`–n`	Norwegian
`–s`	Swedish
`–z`	Swiss
`–x`	This option can be used to convert a file containing IBM characters to a file containing characters from the ISO 8859-1 character set [see `meta-ascii(5)`].

`–o` *outfile*

Output from `sinfilt` is redirected to the output file *outfile*.

file Name of the file to be converted.

file not specified: `sinfilt` reads from standard input.

To convert Reliant UNIX files to MS-DOS files you use the `dosfilt(1)` command.

The commands `sinfilt –x` and `dosfilt –x` convert the specified files from the IBM character set to the ISO 8859-1 character set (`sinfilt`) and vice-versa (`dosfilt`). Since these two character sets do not contain exactly the same range of characters, the following restrictions should be noted:

- 52 characters (including all ASCII characters and German umlauts) can be converted directly on a 1:1 basis.
- All semigraphic characters and a large number of math and Greek characters in the IBM character set (a total of 63 characters with hex codes above 7f) cannot be converted directly to identical characters from the ISO character set. They are replaced by similar characters.
- Some country-specific special characters in the ISO 8859-1 character set (44 characters in all) cannot be directly converted to identical characters in the IBM character set. They are also replaced by similar characters.

EXAMPLES

Converting the MS-DOS file `SCHEDULE` in the IBM character set to the file `schedule`, which can be read under Reliant UNIX.

```
$ sinfilt –x –o schedule SCHEDULE
```

SEE ALSO

dosfilt(1), ascii(5), meta-ascii(5).

User's Reference Manual

sleep(1)

NAME

`sleep -` suspend process execution for an interval

SYNOPSIS

`sleep` *time*

DESCRIPTION

The `sleep` command suspends the execution of the process that calls it for a user-defined period of time.

`sleep` is used mainly in shell scripts to delay execution of the next command.

ARGUMENTS

time The number of seconds after which execution of the process is to resume. At least *time* seconds must elapse.

time must be a non-negative decimal number.

ERROR MESSAGE

`sleep: bad character in argument`

You have used a negative integer or a non-numeric expression for *time*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

This example uses `sleep` from the command line. You start a background process to remind yourself in 10 minutes (600 seconds) that you need to make a phone call.

```
$ (sleep 600; echo 'Call Mr. Jones!')&
```

Example 2

In this example, `sleep` is used within a shell script named `always`, which calls the `backup` program every two minutes:

```
$ cat always
while true
   do
      backup
      sleep 120
   done
```

If you run the `always` script as a background process, you will only be able to terminate it with `kill`, not with the <DEL> key.

SEE ALSO

`wait`(1), `alarm`(2), `sleep`(3C).

User's Reference Manual

slogin(1)

NAME

`slogin` - secure login (remote login program)

SYNOPSIS

`slogin` [`-l` *login_name*] *hostname* [*command*]

`slogin` [`-k`] [`-c` `blowfish|idea|des|3des|none`] [`-e` *ch|^ch|none*] [`-i` *identity_file*] [`-l` *login_name*] [`-n`] [`-o` *option*] [`-p` *port*] [`-q`] [`-t`] [`-v`] [`-x`] [`-X`] [`-C`] [`-L` *port*:*host*:*hostport*] [`-R` *port*:*host*:*hostport*] *hostname* [*command*]

DESCRIPTION

`ssh` (Secure Shell) is a program for logging into a remote machine and for executing commands in a remote machine. It is intended to replace `rlogin` and `rsh` and to provide secure encrypted communication between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

`ssh` connects and logs into the specified *hostname*. The user must prove his/her identity to the remote machine using one of several methods.

Firstly, if the machine the user logs in from is listed in `/etc/hosts.equiv` or `/etc/shosts.equiv` on the remote machine, and the user names are the same on both sides, the user is immediately permitted to log in. Secondly, if `.rhosts` or `.shosts` exists in the user's home directory on the remote machine and contains a line containing the name of the client machine and the name of the user on that machine, the user is permitted to log in. This form of authentication alone is normally not allowed by the server because it is not secure.

The second (and primary) authentication method is the `.rhosts` or `hosts.equiv` method combined with RSA (Rivest-Shamir-Adelman)-based host authentication. It means that if the login is permitted by `.rhosts`, `.shosts`, `/etc/hosts.equiv` or `/etc/shosts.equiv`, and additionally that it can verify the client's host key (see `$HOME/.ssh/known_hosts` and `/etc/ssh_known_hosts` in the **FILES** section), only then is login permitted. This authentication method closes security holes due to IP spoofing, DNS spoofing and routing spoofing. [Note to the administrator: `/etc/hosts.equiv`, `.rhosts` and the `rlogin/rsh` protocol in general are inherently insecure and should be disabled if security is desired.]

As a third authentication method, `slogin` supports RSA based authentication. The scheme is based on public-key cryptography: there are crypto systems where encryption and decryption are performed using separate keys and it is not possible to derive the decryption key from the encryption key. RSA is one such system. The idea is that each user creates a public/private key pair for authentication purposes. The server knows the public key and only the user knows the private key. The `$HOME/.ssh/authorized_keys` file lists the public keys that are permitted for logging in. When the user logs in, the `slogin` program tells the server which key pair it would like to use for authentication. The server checks if this key is permitted and if so sends the user (actually the `slogin` program running on behalf of the user) a challenge, a random number, encrypted by the user's public key. The challenge can only be decrypted using the proper private key. The user's client then decrypts the challenge using the private key, proving that he/she knows the private key but without disclosing it to the server.

`ssh` implements the RSA authentication protocol automatically. The user creates his/her RSA key pair by running `ssh-keygen(1)`. This stores the private key in `.ssh/identity` and the public key in `.ssh/identity.pub` in the user's home directory. The user should then copy the `identity.pub` to `.ssh/authorized_keys` in his/her home directory on the remote machine (the `authorized_keys` file corresponds to the conventional `.rhosts` file, and has one key per line, though the lines can be very long). After this, the user can log in without giving the password. RSA authentication is much more secure than `.rhosts` authentication.

The most convenient way to use RSA authentication may be with an authentication agent. See `ssh-agent(1)` for more information.

If other authentication methods fail, `slogin` prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

When the user's identity has been accepted by the server, the server either executes the given command or logs into the machine and gives the user a normal shell on the remote machine. All communication with the remote command or shell will be automatically encrypted.

If a pseudo-terminal has been allocated (normal login session), the user can disconnect with "~.", and suspend `slogin` with "~^Z". All forwarded connections can be listed with "~#", and if the session blocks waiting for forwarded X11 or TCP/IP connections to terminate, it can be backgrounded with "~&" (this should not be used while the user shell is active, as it can cause the shell to hang). All available escapes can be listed with "~?".

A single tilde character can be sent as "~~" (or by following the tilde with a character other than those described above). The escape character must always follow a newline to be interpreted as special. The escape character can be changed in configuration files or from the command line.

If no pseudo terminal has been allocated, the session is transparent and can be used to reliably transfer binary data. On most systems, setting the escape character to "none" will also make the session transparent even if a tty is used.

The session terminates when the command or shell on the remote machine exits and all X11 and TCP/IP connections have been closed. The exit status of the remote program is returned as the exit status of `slogin`.

If the user is using X11 (the `DISPLAY` environment variable is set), the connection to the X11 display is automatically forwarded to the remote side in such a way that any X11 programs started from the shell (or command) will go through the encrypted channel and the connection to the real X server will be made from the local machine. The user should not manually set `DISPLAY`. Forwarding of X11 connections can be configured from the command line or in configuration files.

The `DISPLAY` value set by `slogin` will point to the server machine, but with a display number greater than zero. This is normal and happens because `slogin` creates a "proxy" X server on the server machine for forwarding the connections over the encrypted channel.

`ssh` will also automatically set up X-authority data on the server machine. For this purpose, it will generate a random authorization cookie, store it in X-authority on the server and verify that any forwarded connections carry this cookie and replace it by the real cookie when the connection is opened. The real authentication cookie is never sent to the server machine (and no cookies are sent in plain text).

If the user is using an authentication agent, the connection to the agent is automatically forwarded to the remote side unless disabled from the command line or in a configuration file.

Forwarding of arbitrary TCP/IP connections over the secure channel can be specified either from the command line or in a configuration file. One possible application of TCP/IP forwarding is a secure connection to an electronic purse; another is going through firewalls.

`ssh` automatically maintains and checks a database containing RSA (Rivest-Shamir-Adelman) based identifications for all hosts it has ever been used with. The database is stored in `.ssh/known_hosts` in the user's home directory. Additionally, the file `/etc/ssh_known_hosts` is automatically checked for known hosts. Any new hosts are automatically added to the user's file. If a host's identification ever changes, `slogin` warns about this and disables password authentication to prevent a Trojan horse from getting the user's password. Another purpose of this mechanism is to prevent "man-in-the-middle" attacks (for details see: `http://www.ssh.com/tech/crypto/intro.html#timing-attack`) which could otherwise be used to circumvent the encryption. The `StrictHostKeyChecking` option (see below) can be used to prevent logins to machines whose host key is not known or has changed.

OPTIONS

`-c blowfish|idea|des|3des|none`

Selects the cipher to use for encrypting the session.

`blowfish`

Blowfish is an algorithm developed by Bruce Schneier. It is a block cipher with 64-bit block size

and variable length keys (up to 448 bits). It has gained a fair amount of acceptance in a number of applications. No attacks are known against it.

`idea` IDEA (International Data Encryption Algorithm) is an algorithm developed at ETH Zurich in Switzerland. It uses a 128 bit key and is generally considered to be very secure. It is currently one of the best known public algorithms. It is a fairly new algorithm, but it has already been around for several years. No practical attacks on it have been published despite numerous attempts to analyze it.

`des` DES is the data encryption standard, but is breakable by governments, large corporations and major criminal organizations.

`3des` 3DES (triple-des) is based on using `des` three times in an encrypt-decrypt-encrypt sequence with three different, unrelated keys. It is presumably more secure than `des`.

`3des` is used by default.

`none` Disables encryption entirely; it is only intended for debugging and it renders the connection insecure.

For more information see `http://www.ssh.com/tech/crypto/algorithms.html`.

`-e` *ch*|`^`*ch*|`none`
Sets the escape character for sessions with a pseudo-terminal to *ch* or `^`*ch* (default: ~). The escape character is only recognized at the beginning of a line. The escape character followed by a dot (.) closes the connection, followed by control-Z suspends the connection and followed by itself sends the escape character once. Setting the character to `none` disables any escapes and makes the session fully transparent.

`-f` Requests `slogin` to go to background after authentication. This is useful if `slogin` is going to prompt for passwords or passphrases, but the user wants it in the background. This implies `-n`. The recommended way to start X11 programs at a remote site is with something like "`slogin -f host xterm`".

`-i` *identity_file*
Selects the file from which the identity (private key) for RSA authentication is read. Default is `.ssh/identity` in the user's home directory. Identity files may also be specified on a per-host basis in the configuration file. It is possible to have multiple `-i` options (and multiple identities specified in configuration files).

`-k` Disables forwarding of Kerberos tickets/AFS tokens. This may also be specified on a per-host basis in the configuration file.

`-l` *login_name*
Specifies the user to log in as on the remote machine. This may also be specified on a per-host basis in the configuration file.

`-n` Redirects stdin from `/dev/null` (actually, prevents reading from stdin). This must be used when `slogin` is run in the background. A common trick is to use this to run X11 programs on a remote machine. For example, "`slogin -n shadows.cs.hut.fi emacs &`" will start an `emacs` on `shadows.cs.hut.fi` and the X11 connection will be automatically forwarded over an encrypted channel. The `slogin` program will be put in the background. (This does not work if `slogin` needs to prompt for a password or passphrase; see also the `-f` option.)

`-o` *option*
Can be used to give options in the format used in the configuration file. This is useful for specifying options for which there is no separate command-line flag. The option has the same format as a line in the configuration file.

`-p` *port*
Port to connect to on the remote host. This can be specified on a per-host basis in the configuration file.

`-q` Quiet mode. Causes all warning and diagnostic messages to be suppressed. Only fatal errors are

displayed.

`-t` Force pseudo-terminal allocation. This can be used to execute arbitrary screen-based programs on a remote machine, which can be very useful, for example, when implementing menu services.

`-v` Verbose mode. Causes `slogin` to print debugging messages about its progress. This is helpful in debugging connection, authentication and configuration problems.

`-x` Disables X11 forwarding. This can also be specified on a per-host basis in a configuration file.

`-X` Enables X11 forwarding.

`-C` Requests compression of all data (including stdin, stdout, stderr, and data for forwarded X11 and TCP/IP connections). The compression algorithm is the same used by `gzip` and the "level" can be controlled by the `CompressionLevel` option (see below). Compression is desirable on modem lines and other slow connections, but will only slow down things on fast networks. The default value can be set on a host-by-host basis in the configuration files; see the `Compress` option below.

`-L` *port*:*host*:*hostport*
Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to *port* on the local side. Then whenever a connection is made to this port, the connection is forwarded over the secure channel and a connection is made to *host*:*hostport* from the remote machine. Port forwardings can also be specified in the configuration file. Only root can forward privileged ports.

`-R` *port*:*host*:*hostport*
Specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side. This works by allocating a socket to listen to *port* on the remote side. Then whenever a connection is made to this port, the connection is forwarded over the secure channel and a connection is made to *host*:*hostport* from the local machine. Port forwardings can also be specified in the configuration file. Privileged ports can be forwarded only when logging in as root on the remote machine.

CONFIGURATION FILES

`ssh` obtains configuration data from the following sources (in this order): command line options, user's configuration file (`$HOME/.ssh/config`), and system-wide configuration file (`/etc/ssh_config`). For each parameter, the first obtained value will be used. The configuration files contain sections bracketed by "Host" specifications, and that section is only applied for hosts that match one of the patterns given in the specification. The matched host name is the one given at the command line.

Since the first obtained value for each parameter is used, more host-specific declarations should be given near the beginning of the file, with general defaults at the end.

The configuration file has the following format:

Empty lines and lines starting with "#" are comments.

Otherwise a line is of the format "keyword arguments". The possible keywords and their meanings are as follows (note that the configuration files are case-sensitive):

`Host` Restricts the following declarations (up to the next `Host` keyword) to be only for those hosts that match one of the patterns given after the keyword. "*" and "?" can be used as wildcards in the patterns. A single "*" as a pattern can be used to provide global defaults for all hosts. The host is the *hostname* argument given at the command line (i.e. the name is not converted to a canonical host name before matching).

`AFSTokenPassing`
Specifies whether to pass AFS tokens to remote hosts. The argument to this keyword must be "`yes`" or "`no`".

`BatchMode`
If set to "`yes`", passphrase/password querying will be disabled. This option is useful in scripts and other batch jobs where you have no user to supply the password. The argument must be "`yes`" or "`no`".

`Cipher`

Specifies the cipher to use for encrypting the session. Currently, `blowfish`, `idea`, `des`, `3des` and `none` are supported. The default is `3des`. Using `none` (no encryption) is intended only for debugging and will render the connection insecure (cf. `-c` option above).

`Compression`
Specifies whether to use compression. The argument must be "`yes`" or "`no`".

`CompressionLevel`
Specifies the compression level to use if compression is enabled. The argument must be an integer from 1 (fast) to 9 (slow, best). The default level is 6, which is good for most applications. The meaning of the values is the same as in GNU GZIP.

`ConnectionAttempts`
Specifies the number of tries (one per second) to make before falling back to `rsh` or exiting. The argument must be an integer. This may be useful in scripts if the connection sometimes fails.

`EscapeChar`
Sets the escape character (default: ~). The escape character can also be set at the command line. The argument should be a single character, "^" followed by a letter, or `none` to disable the escape character entirely (making the connection transparent for binary data).

`FallBackToRsh`
Specifies that if connecting via `slogin` fails due to a connection refused error (there is no `sshd` listening on the remote host), `rsh` should automatically be used instead (after a suitable warning about the session being unencrypted). The argument must be "`yes`" or "`no`".

`ForwardAgent`
Specifies whether the connection to the authentication agent (if any) will be forwarded to the remote machine. The argument must be "`yes`" or "`no`".

`ForwardX11`
Specifies whether X11 connections will be automatically redirected over the secure channel and `DISPLAY` set. The argument must be "`yes`" or "`no`".

`GlobalKnownHostsFile`
Specifies a file to use instead of `/etc/ssh_known_hosts`.

`HostName`
Specifies the real host name to log into. This can be used to specify nicknames or abbreviations for hosts. The default is the name given at the command line. Numeric IP addresses are also permitted (both at the command line and in `HostName` specifications).

`IdentityFile`
Specifies the file from which the user's RSA authentication identity is read (default `.ssh/identity` in the user's home directory). Additionally, any identities represented by the authentication agent will be used for authentication. The file name may use the tilde syntax to refer to a user's home directory. It is possible to have multiple identity files specified in configuration files; all these identities will be tried in sequence.

`KeepAlive`
Specifies whether the system should send "keepalive" messages to the other side. If they are sent, the death of the connection or crash of one of the machines will be properly noticed. However, this means that connections will die if the route is down temporarily and some people find this annoying.

The default is "`yes`" (to send keepalives), and the client will notice if the network goes down or the remote host dies. This is important in scripts and many users want it too.

To disable keepalives, the value should be set to "`no`" in both the server and the client configuration files.

`KerberosAuthentication`
Specifies whether Kerberos authentication will be used.

`KerberosTgtPassing`
Specifies whether a Kerberos TGT will be forwarded to the server. Note that TGT forwarding is normally not enabled in the server.

`LocalForward`
: Specifies that a TCP/IP port on the local machine be forwarded over the secure channel to the given *host*`:`*port* on the remote machine. The first argument must be a port number and the second must be *host*`:`*port*. Multiple forwardings may be specified and additional forwardings can be given at the command line. Only root can forward privileged ports.

`PasswordAuthentication`
: Specifies whether to use password authentication. The argument to this keyword must be "`yes`" or "`no`".

`Port` Specifies the port number to connect on the remote host. Default is 22.

`ProxyCommand`
: Specifies the command to use to connect to the server. The command string extends to the end of the line and is executed with `/bin/sh`. In the command string, `%h` will be substituted by the host name to connect and `%p` by the port. The command can be basically anything and should read from its stdin and write to its stdout. It should eventually connect an `sshd` server running on some machine or execute "`sshd –i`" somewhere. Host key management will be performed using the `HostName` of the host being connected (defaulting to the name typed by the user).

  Note that `slogin` can also be configured to support the `SOCKS` system using the `--with-socks` compile time configuration option.

`RemoteForward`
: Specifies that a TCP/IP port on the remote machine be forwarded over the secure channel to the given *host*`:`*port* on the local machine. The first argument must be a port number and the second must be *host*`:`*port*. Multiple forwardings may be specified and additional forwardings can be given at the command line. Only root can forward privileged ports.

`RhostsAuthentication`
: Specifies whether to try `.rhosts` based authentication. Note that this declaration only affects the client side and has no effect whatsoever on security. Disabling `.rhosts` authentication may reduce authentication time on slow connections when `.rhosts` authentication is not used. Most servers do not permit `RhostsAuthentication` because it is not secure (see `RhostsRSAAuthentication`). The argument to this keyword must be "`yes`" or "`no`".

`RhostsRSAAuthentication`
: Specifies whether to try `.rhosts` based authentication with RSA host authentication. This is the primary authentication method for most sites. The argument must be "`yes`" or "`no`".

`RSAAuthentication`
: Specifies whether to try RSA authentication. The argument to this keyword must be "`yes`" or "`no`". RSA authentication will only be attempted if the identity file exists or an authentication agent is running.

`StrictHostKeyChecking`
: If this flag is set to "`yes`", `slogin` will never automatically add host keys to the `$HOME/.ssh/known_hosts` file and refuses to connect hosts whose host key has changed. This provides maximum protection against Trojan horse attacks. However, it can be somewhat annoying if you do not have good `/etc/ssh_known_hosts` files installed and frequently connect new hosts. Basically this option forces the user to manually add any new hosts. Normally this option is disabled and new hosts will automatically be added to the known host files. The host keys of known hosts will be verified automatically in either case. The argument must be "`yes`" or "`no`".

`User` Specifies the user to log in as. This can be useful if you have a different user name on different machines. This saves the trouble of having to remember to enter the user name at the command line.

`UserKnownHostsFile`
: Specifies a file to use instead of `$HOME/.ssh/known_hosts`.

`UseRsh`
: Specifies that `rlogin/rsh` should be used for this host. It is possible that the host does not support the `slogin` protocol. This causes `slogin` to immediately execute `rsh`. All other options (except `HostName`) are ignored if this has been specified. The argument must be "`yes`" or "`no`".

ENVIRONMENT

`ssh` will normally set the following environment variables:

`DISPLAY`
: The `DISPLAY` variable indicates the location of the X11 server. It is automatically set by `slogin` to point to a value of the form "*hostname*`:`*n*" where *hostname* indicates the host where the shell runs and *n* is an integer ≥ 1. `ssh` uses this special value to forward X11 connections over the secure channel. The user should normally not set `DISPLAY` explicitly, as that will render the X11 connection insecure (and will require the user to manually copy any required authorization cookies).

`HOME`
: Set to the path of the user's home directory.

`LOGNAME`
: Synonym for `USER`; set for compatibility with systems that use this variable.

`MAIL`
: Set to point the user's mailbox.

`PATH`
: Set to the default `PATH`, as specified when compiling `slogin` or `/etc/default/login`.

`SSH_AUTHENTICATION_FD`
: This is set to an integer value if you are using the authentication agent and a connection to it has been forwarded. The value indicates a file descriptor number used for communicating with the agent. On some systems, `SSH_AUTHENTICATION_SOCKET` may be used instead to indicate the path of a Unix domain socket used to communicate with the agent (this method is less secure and is only used on systems that do not support the first method).

`SSH_CLIENT`
: Identifies the client end of the connection. The variable contains three space-separated values: client ip address, client port number and server port number.

`SSH_TTY`
: This is set to the name of the tty (path to the device) associated with the current shell or command. If the current session has no tty, this variable is not set.

`TZ`
: The timezone variable is set to indicate the present timezone if it was set when the daemon was started (i.e. the daemon passes the value on to new connections).

`USER`
: Set to the name of the user logging in.

Additionally, `slogin` reads `/etc/default/login` and `$HOME/.ssh/environment` and adds lines of the format `VARNAME=`*value* to the environment.

FILES

`$HOME/.ssh/known_hosts`
: Records host keys for all hosts the user has logged into (that are not in `/etc/ssh_known_hosts`). See `sshd(8)` .

`$HOME/.ssh/random_seed`
: Used for seeding the random number generator. This file contains sensitive data and should be read/write for the user and not accessible for others. This file is created the first time the program is run and updated automatically. The user should never need to read or modify this file.

`$HOME/.ssh/identity`
: Contains the RSA authentication identity of the user. This file contains sensitive data and should be readable by the user but not accessible by others. It is possible to specify a passphrase when generating the key; the passphrase will be used to encrypt the sensitive part of this file using `IDEA` (cf. `-c` option above).

`$HOME/.ssh/identity.pub`
: Contains the public key for authentication (public part of the identity file in human-readable form). The contents of this file should be added to `$HOME/.ssh/authorized_keys` on all machines where you wish to log in using RSA authentication. This file is not sensitive and can (but need not) be readable by anyone. This file is never used automatically and is not necessary; it is only provided for the convenience of the user.

`$HOME/.ssh/config`
: This is the per-user configuration file. The format of this file is described above. This file is used by the `slogin` client. This file does not usually contain any sensitive information, but the recommended permissions are read/write for the user and not accessible by others.

`$HOME/.ssh/authorized_keys`
: Lists the RSA keys that can be used for logging in as this user. The format of this file is described in the `sshd(8)` manual page. In its simplest form, the format is the same as the `\.pub` identity files (that is, each line contains the number of bits in modulus, public exponent, modulus and comment fields, separated by spaces). This file is not highly sensitive, but the recommended permissions are read/write for the user and not accessible by others.

`/etc/ssh_known_hosts`
: System-wide list of known host keys. This file should be prepared by the system administrator to contain the public host keys of all machines in the organization. This file should be world-readable. This file contains public keys, one per line, in the following format (fields separated by spaces): system name, number of bits in modulus, public exponent, modulus and optional comment field. When different names are used for the same machine, all such names should be listed, separated by commas. The format is described on the `sshd(8)` manual page.

  The canonical system name (as returned by name servers) is used by `sshd` to verify the client host when logging in; other names are needed because `slogin` does not convert the user-supplied name to a canonical name before checking the key, because someone with access to the name servers would then be able to fool host authentication.

`/etc/ssh_config`
: System-wide configuration file. This file provides defaults for those values that are not specified in the user's configuration file and for those users who do not have a configuration file. This file must be world-readable.

`$HOME/.rhosts`
: This file is used in `.rhosts` authentication to list the host/user pairs that are permitted to log in. (Note that this file is also used by `rlogin` and `rsh`, which makes using this file insecure.) Each line in the file contains a host name (in the canonical form returned by name servers) and then a user name on that host, separated by a space. On some machines, this file may need to be world-readable if the user's home directory is on a NFS partition, because `sshd` reads it as root. Additionally, this file must be owned by the user and must not have write permissions for anyone else. The recommended permission for most machines is read/write for the user and not accessible by others.

  Note that by default `sshd` will be installed so that it requires successful RSA host authentication before permitting `.rhosts` authentication. If your server machine does not have the client's host key in `/etc/ssh_known_hosts`, you can store it in `$HOME/.ssh/known_hosts`. The easiest way to do this is to connect back to the client from the server machine using `ssh`; this will automatically add the host key in `$HOME/.ssh/known_hosts`.

`$HOME/.shosts`
: This file is used in exactly the same way as `.rhosts`. The purpose of having this file is to be able to use `.rhosts` authentication with `slogin` without permitting login with `rlogin` or `rsh`.

`/etc/hosts.equiv`
: This file is used during `.rhosts` authentication. It contains canonical hosts names, one per line (the full format is described on the `sshd(8)` manual page). If the client host is found in this file, login is automatically permitted provided client and server user names are the same. Additionally, successful RSA host authentication is normally required. This file should only be writeable by root.

`/etc/shosts.equiv`
: This file is processed exactly as `/etc/hosts.equiv`. This file may be useful to permit logins using `slogin` but not using `rsh/rlogin`.

`/etc/sshrc`
: Commands in this file are executed by `slogin` when the user logs in, just before the user's shell (or command) is started. See the `sshd(8)` manual page for more information.

`$HOME/.ssh/rc`
: Commands in this file are executed by `slogin` when the user logs in. just before the user's shell (or command) is started. See the `sshd(8)` manual page for more information.

INSTALLATION

`ssh` is normally installed as suid root. It needs root privileges only for `.rhosts` authentication (`.rhosts` authentication requires that the connection must come from a privileged port and allocating such a port requires root privileges). It also needs to be able to read `/etc/ssh_host_key` to perform RSA host authentication. It is possible to use `slogin` without root privileges, but `.rhosts` authentication will then be disabled. `ssh` drops any extra privileges immediately after the connection to the remote host has been made.

REFERENCES

`http://www.ssh.com`

`http://www.ssh.org`

SEE ALSO

`rlogin(1)`, `rsh(1)`, `scp(1)`, `ssh-add(1)`, `ssh-agent(1)`, `ssh-keygen(1)`, `telnet(1)`, `sshd(8)`
.

User's Reference Manual

soelim(1)
BSD Compatibility Package

NAME

`soelim -` resolve and eliminate .so requests from nroff or troffinput

SYNOPSIS

`/usr/ucb/soelim` [*filename* ...]

DESCRIPTION

The `soelim` command reads the specified files or standard input and includes the text specified by the `nroff`(1M) directives in the form

`.so` *somefile*

when they appear at the beginning of input lines. This is useful since programs such as `tbl`(1M) do not normally do this; it allows the placement of individual tables in separate files to be run as part of a large document.

An argument consisting of "–" is taken to be a file name corresponding to standard input.

EXAMPLE

A sample usage of `soelim` would be

```
soelim exum?.n | tbl | nroff –ms | col | lpr
```

SEE ALSO

`nroff`(1M), `tbl`(1M).

User's Reference Manual

sort(1)

NAME

`sort` - sort and/or merge files

SYNOPSIS

`sort` [*option ...*] [*file ...*]

DESCRIPTION

`sort` sorts lines in an input file and writes the result to standard output.

If you specify more than one file, `sort` sorts and merges the files in the same operation, i.e. the contents of all input files are sorted and printed together.

Sorting can be performed either by whole lines or by specific parts of lines, known as sort keys. If you wish to sort by whole lines, you do not specify any sort keys; one or more keys can be used to sort by particular portions of lines. A sort key is defined by specifying the positions of fields in a line in the form *+pos1 –pos2* (see **Defining specific sort keys**).

`sort` divides the lines of a file into fields. A field is a string of characters that is delimited by a field separator or a newline. Blanks and tabs are the default field separators. In a sequence of one or more default separators, all separators are part of the next field. Leading blanks at the beginning of a line thus by default form part of the first field.

OPTIONS

No option specified

`sort` sorts the input lines lexicographically, whereby a byte is used for each single character. The sorting order defined using `LC_COLLATE` is valid for the characters.

Options that alter the behavior of sort

`–c` `sort` checks whether the input file is already sorted according to the current ordering rules. If it is, nothing is output; otherwise, the first line that does not match the ordering rules is displayed.

Only one file may be specified with option `–c`! The options `–m` and `–o` must not be combined with `–c`.

Together with `–u`: `sort` also checks whether there are lines with identical sort keys available.

`–m` `sort` merges input files that are already sorted.

`–m` must not be combined with `–c`.

`–o` *output_file*

output_file is the name of a file to which the sorted contents of the input file are to be written. The file named as *output_file* can also be one of the input files, but in this case the original unsorted contents of the named file are overwritten.

Only one `–o` option must be specified. `–o` must not be combined with `–c`.

`–o` *output_file* not specified: `sort` writes to standard output.

`–T` *directory*

Specifies a *directory* for temporary files.

`–T` not specified: Temporary files are created in `/var/tmp`.

`–u` (unique) Causes identical lines to be output once only. Lines with identical sort keys are considered identical lines.

`–y` [*kmem*]

Option `–y` defines the memory size that `sort` uses to start with. This initial size has a large impact on the speed with which the file is sorted. It is a waste of memory or of CPU time to sort a small file in a large amount of memory or a large file in a small amount of memory respectively.

kmem Amount of memory (in Kbytes) initially assigned to `sort`. If you assign a value above the maximum of 1 Mbyte or below the minimum of 16 Kbyte, the corresponding extremum will be used. Thus if you define a value of 0 (`-y0`), for example, `sort` will start with minimum memory.

kmem not specified: `sort` starts with maximum memory.

`-y` [*kmem*] not specified:

`sort` starts with a system default memory size (32 Kbytes), and continues to use more space if required.

`-z` *recsz*

With this option you allocate correctly sized buffers for the merge phase. You only need to do this if you are using option `-c` or `-m`, i.e. if you are not actually sorting the files.

If you are sorting the files, `sort` records the size of the longest line read in the sort phase so that buffers of the correct size can be allocated during the merge phase.

If you are not sorting the files, `sort` normally uses a default value for the buffer size. Lines longer than this will cause `sort` to terminate abnormally. Supplying the actual number of bytes in the longest line to be merged (or some larger value) will prevent abnormal termination.

Options that alter ordering rules

The following options can be specified in either of two ways:

- either as options before the first positioning specification:

  They are then valid globally for all subsequent sort keys. When using `-k`, the options must be placed before the first specification of `-k`; in the case of *+pos* or *-pos*, the options can also be placed between the additional positioning specifications, and are then only valid for the subsequent sort keys.

- or as modifiers for individual sort keys:

  You then cancel the global settings for the relevant sorting field, i.e. a change to the ordering rule is only valid when made in accordance with the specified modification.

  Option letters appended to the field specification without a dash or a blank act as modifiers (see **Defining specific sort keys**).

`-b` Ignores leading field separators when determining the start and end of a sort key. Note that the `b` option is only effective when sorting is based on sort keys (i.e. not on the whole line).

`-d` Performs a lexicographical sort, taking into account only the characters for which the C functions `isalnum(3C)` and `isspace(3C)` return a value of "true". These are the characters defined in the current locale as alphanumeric letters, digits, or characters producing white space, such as blanks or tabs.

`-f` Folds lowercase into uppercase before sorting, thus making no distinction between them.

`-i` In non-numeric comparisons, ignores all characters for which the C function `isprint(3C)` returns a value of "false", i.e. all characters defined as non-printing in the current locale. If the collating sequence is based on the ASCII table, for example, characters 001 through 037 (octal) and character 0177 (octal) are ignored [see `ascii(5)`].

`-M` The first three characters of the sort key are converted to uppercase, treated as names of months, and collated in calendar order. The `-M` option implies the `-b` option.

`-n` Sorts numerically. A numeric value must come first in the sort key and may consist of: blanks, minus signs, digits 0-9, and a decimal point. The `-n` option implies the `-b` option, i.e. leading blanks are ignored.

`-r` Reverses the collating sequence (sorting order).

Option that alters field separators

`-t` *x* Uses the character you specify for *x* as the field separator. Unlike default field separators, *x* is itself not part of a field. It may, however, be part of a sort key, for example if the sort key extends from the first to

the third *x*-separated field. Every field separator *x* is significant, i.e. *xx* delimits an empty field.

`–t` not specified:

The default field separators apply (blanks and tabs). A sequence of one or more default field separators forms part of the following field.

Defining specific sort keys

When defining sort keys please note that sequences of letters defined as one collating element in the current locale count as a single letter. In a Spanish locale, for example, `ch` is a single collating element.

Specifying sort keys with the new synopsis `–k` *field_separator* has the same effect as using the old synopsis *+pos1* or *–pos2*, but the two must not be combined. Conversion to the new synopsis is recommended.

You can specify several sort keys. Where there are several sort keys, `sort` first sorts by the first sort key, moves on to the next if the first sort key is equal, and so on.

`–k` *field_separator*

With `–k` you define start and end of a sort key. In *field_separator* you define the first and last character of the sort key.

field_separator has the following format:

start_field[*type*][,*end_field*[*type*]]

whereby the number of the field and a character in the field can be specified for *start_field* and *end_field*:

m[.*n*]

m and *n* are integers with the following significance:

m *m* specifies the number of the first or last field.

.*n* *n* specifies the number of the first character used in the first field or the number of the last character used in the last field.

n not specified:

The field is used by the first character through to the last character.

type modifies the sort key (see **Options that alter ordering rules**).

+pos1 [*–pos2*]

+pos1 and *–pos2* specify the start and end of a sort key on the basis of the fields in the input lines.

+pos1

is the position of the first character *in* the sort key;

–pos2

refers to the first character *after* it. *+pos1* must come before *–pos2*.

–pos2 not specified:

The sort key extends from *+pos1* to the end of the line.

The *pos1* and *pos2* arguments have the form:

m[.*n*][*type*]

where *m* and *n* are integers with the following significance:

m Skips *m* fields of the line, addressing field *m*+1.

.*n* Skips *n* characters plus the field separator as of the last character of field *m*, thus addressing character *n*+1 within field *m*+1. If the `–b` option is in effect, field separators at the start of a field are not counted; thus, *+m*.*n*`b` refers to the *n*+1th non-whitespace character after field *m*.

.*n* not specified:

Is equivalent to `.0` and refers to the first character after field *m*. If the `–b` option is in effect, field separators at the start of a field are not counted; thus, *+m*`.0b` refers to the first non-whitespace character in the *m*+1th field.

type Modifies the sort key (see **Options that alter ordering rules**).

Example:

To specify a sort key that begins with the fourth character in the second field and ends with this field, you enter:

`sort -k 2.4,2` (new synopsis) or

`sort +1.3 -2` (old synopsis)

Explanation:

```
                End          End      End
                Field1       Field2   Field3
                     |            |        |
          030-456537 A.Mackenzie   Dublin
                           |       |
                           Sort key:
```

2.4 Start at the 4th character of the 2nd field

+1.3 Skip field 1 and 3 characters:

the 4th character after field 1 is the 1st character in the sort key: M

-2 Skip field 2 and 0 characters:

the 1st character after field 2 is the 1st character after the sort field: blank. Thus the character before is the last character in the sort key: n

Note that default field separators, unlike those defined with option `-t`, are part of the following field. Hence the first character of field 2 is the blank, the second character is the A, and so on.

`--` End of the list of options. Must be specified if *file* begins with `-`.

file Name of the file you wish to sort.

You may name more than one file. All named files are sorted and merged, and the input lines from all of them together are sorted and written to standard output. In the input files, any letter sequence defined as a collating element in the current locale counts as a single letter. Thus in a Spanish locale `ch` is a single collating element. When the last line in an input file is missing a newline character, `sort` appends one, issues a warning, and continues.

Only one file may be specified together with the `-c` option.

If you use a dash (`-`) as the name for *file*, `sort` reads from standard input.

file not specified: `sort` reads from standard input.

EXIT STATUS

`0` All input files were processed correctly. The input file was sorted correctly when `-c` was specified.

`1` `-c` specified: The input file was not sorted correctly. `-c -u` specified: Lines of input with identical sort keys were found.

`>1` Error

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

`LC_COLLATE` governs the preset collating sequence used by the `sort` command.

`LC_CTYPE` governs how character classes are handled by the `-b`, `-d`, `-f` and `-i` options.

`LC_NUMERIC` governs the form of the radix character (decimal point) in conjunction with the `-n` option.

`LC_TIME` governs the currently valid month names, their abbreviations and their collating sequence in conjunction with option `-M`.

Answers to yes/no queries must be given in the language appropriate to the current locale.

If `LC_MESSAGES`, `LC_COLLATE`, `LC_CTYPE`, `LC_NUMERIC` or `LC_TIME` is undefined or is defined as the null

string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Sorting the contents of input_file with the second field as the sort key.

```
$ sort –k 2,2 input_file
```

Example 2

Sorting the contents of input_file1 and input_file2 in reverse order, placing the output in output_file, and using the first character in the second field as the sort key.

```
$ sort –r –o output_file –k 2.1,2.2 input_file1 input_file2
```

Example 3

Sort the contents of input_file1 and input_file2 in reverse order, using the first non-blank character of the second field as the sort key.

```
$ sort –r –o output_file –k 2.0b,2.1b input_file1 input_file2
```

Example 4

Displaying the /etc/passwd file, sorted by the numeric user ID (field 3).

```
$ sort –t : –k 3n,3  /etc/passwd
```

Example 5

Displaying the presorted file input_file, suppressing all but the first occurrence of lines having the same third field.

```
$ sort –u –k 3,3 input_file
```

FILES

/var/tmp/stm???
: Temporary files

SEE ALSO

comm(1), join(1), uniq(1), ctype(3C).

User's Reference Manual

sortbib(1)
BSD Compatibility Package

NAME

`sortbib` - sort a bibliographic database

SYNOPSIS

`/usr/ucb/sortbib` [`–s`*keyletters*] *database* ...

DESCRIPTION

The `sortbib` command sorts files of records containing `refer` keyletters by user-specified keys. Records may be separated by blank lines, or by "`.[`" and "`.]`" delimiters, but the two styles may not be mixed together. This program reads through each *database* and pulls out key fields, which are sorted separately. The sorted key fields contain the file pointer, byte offset, and length of corresponding records. These records are delivered using disk seeks and reads, so `sortbib` may not be used in a pipeline to read standard input.

By default, `sortbib` alphabetizes by the first `%A` and the `%D` fields, which contain the senior author and date. The `–s` option is used to specify new *keyletters*. See `addbib` for a list of the most common key letters. For instance, `–sATD` will sort by author, title, and date, while `–sA+D` will sort by all authors, and date. Sort keys past the fourth are not meaningful. No more than 16 databases may be sorted together at one time. Records longer than 4096 characters will be truncated.

`sortbib` sorts on the last word on the `%A` line, which is assumed to be the author's last name. A word in the final position, such as "`jr.`" or "`ed.`", will be ignored if the name beforehand ends with a comma. Authors with two-word last names or unusual constructions can be sorted correctly by using the `nroff` convention `\0` in place of a blank. A `%Q` field is considered to be the same as `%A`, except sorting begins with the first, not the last, word. `sortbib` sorts on the last word of the `%D` line, usually the year. It also ignores leading articles (like `A` or `The`) when sorting by titles in the `%T` or `%J` fields; it will ignore articles of any modern European language. If a sort-significant field is absent from a record, `sortbib` places that record before other records containing that field.

NOTES

Records with missing author fields should probably be sorted by title.

SEE ALSO

`addbib`(1), `indxbib`(1), `lookbib`(1), `refer`(1), `roffbib`(1).

User's Reference Manual

spell(1)

NAME

`spell` - find spelling errors

SYNOPSIS

`spell` [*option ...*] [*+local_file*] [*file ...*]

DESCRIPTION

`spell` checks the spelling of words in an input file against a spelling list. Words that are not present in the list and which cannot be derived by using standard English grammar rules (e.g. by adding suitable prefixes, suffixes or inflections) are written to standard output. If you wish, you can supplement the default English spelling list with a word list of some other language.

OPTIONS

Caution:

Each individual option must be provided with a minus sign, thus for example, `spell –i –x –v –b`, and not `spell –ixvb`!

`–b` Checks words from *file* for British spelling. Besides preferring `centre`, `colour`, `programme`, `speciality`, `travelled`, etc., this option considers `-ise` in words such as `standardise` to be the correct spelling.

`-b` not specified:

Words from *file* are checked for correct American spelling.

`–i` The `troff` text insertion requests `.so` and `.nx` are always ignored.

`–l` Forces `spell` to follow the chains of all files included as a result of the `troff` text insertion requests `.so` and `.nx`.

If `–l` is not specified, `spell` follows chains of included files only if the path names of such files do not begin with `/usr/lib`.

`–v` Prints all words from *file* that are not literally in the spelling list. The spelling list is located in the `/usr/ucblib/dict/words` file.

Plausible derivations from the words in the spelling list are identified as follows:

`+prefix word`

`+suffix word`

`–x` Every plausible stem is printed with = for each word in *file*.

+local_file

Words found in *local_file* are removed from `spell`'s output. `spell` considers the words in this file to have acceptable spellings. *local_file* is a user-provided file containing a list of words sorted by ASCII codes (one word per line). You will need to have read permission for *local_file* [see `chmod(1)`].

+local_file not specified:

`spell` only uses the spellings in `/usr/ucblib/dict/words`.

file Name of the file to be checked for spelling errors. Read permission is required for *file* [see `chmod(1)`].

file not specified:

`spell` reads from standard input.

`spell` ignores most `troff`, `tbl`, and `eqn` constructions in the input.

ERROR MESSAGES

`spell cannot read` *local_file*

Either the named *local_file* does not exist, or you do not have write permission for it.

ENVIRONMENT VARIABLES

Alternate auxiliary files (spelling lists, stop list, history file) may be specified by using these environment variables. The default values are as follows:

`D_SPELL=/usr/share/lib/spell/hlist[ab]`
: Contains the hashed spelling list.

`S_SPELL=/usr/share/lib/spell/hstop`
: Contains the hashed stop list.

`H_SPELL=/var/adm/spellhist`
: Contains the history file.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

On the basis of the words in `list`, check the file `colors` for misspelled words:

```
$ cat colors
bleu yellow redd grien purple black wite orrange
$ cat list
black
blue
green
orange
purple
red
white
yellow
$ spell -v +list colors
bleu
grien
orrange
redd
wite
```

The output shows all words in `colors` that are not in `list`.

FILES

`/usr/lib/spell/spellprog`
: Executable program used by `spell`.

`/usr/share/lib/spell/hlist[ab]`
: Hashed spelling lists, American and British. The spelling lists are based on many sources, and while less systematic than an ordinary dictionary, is also more effective with respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is limited.

`/usr/share/lib/spell/hstop`
: Hashed stop list. The stop list filters out misspellings (e.g. thier=thy-y+ier) that would otherwise pass.

`/usr/ucblib/dict/words`
: Default spelling list used when no *local_file* is specified.

`/var/adm/spellhist`
: History file in which copies of all output are accumulated.

The above files can be extended and maintained with the commands `spellin`, `hashmake` and `hashcheck` (see the corresponding descriptions).

SEE ALSO

`deroff`(1), `hashmake`(1), `hashcheck`(1), `spellin`(1), `sed`(1), `sort`(1), `tee`(1), `eqn`(1M), `nroff`(1M), `tbl`(1M), `troff`(1M).

User's Reference Manual

spellin(1)

NAME

`spellin -` create compressed spelling list

SYNOPSIS

`/usr/lib/spell/spellin` *count*

DESCRIPTION

`hashmake`, `hashcheck` and `spellin` are a group of auxiliary routines for the spell-checking program `spell`. The `spell` command uses spelling lists to determine the correct spelling of words. You can use the auxiliary routines to maintain and extend the existing spelling lists and to create new lists.

`spellin` creates compressed spelling lists from the specified hash codes. These spelling lists can be used by the `spell` command. `spellin` can also be used to add words to an existing spelling list.

`spellin` reads *count* hash codes (as created by `hashmake`) from standard input, creates a compressed spelling list from them, and writes it to standard output.

`spellin` can be used to create a new spelling list or to add words to an existing spelling list.

DIAGNOSTICS

"`spellin: arg count`"
No *count* was specified

SEE ALSO

`hashcheck`(1), `hashmake`(1), `spell`(1).

User's Reference Manual

split(1)

NAME

`split -` split a file into pieces

SYNOPSIS

`split` [*option ...*] [*file* [*name*]]

DESCRIPTION

`split` divides a file into smaller segments, storing the segments in individual output files and leaving the original file unaltered. The output files are automatically numbered; `split` uses a suffix comprising two lowercase letters (aa, ab ... zz) from the current internationalized environment for this purpose. The last file contains the remainder of the input file and may contain fewer lines than the number specified.

If the number of output files required exceeds the maximum length allowed by the suffix, `split` does not write the last file (as this would contain more lines than specified) and terminates with the exit status >0. The files that have already been created are not deleted.

OPTIONS

No option specified:

The output files are called `xaa`, `xab` etc. up to `xzz` in lexicographical sequence. In this case, `split` creates a maximum of 676 output files.

`–a` *number*

The suffix for the output file consists of *number* letters. For example, `–a 4` creates the output files `xaaaa`, `xaaab` etc. up to `xzzzz`.

`–a` not specified: The suffix consists of 2 letters.

`–b` *bytes*

`split` splits the input file into sections of size *bytes*. *bytes* can be specified as follows:

n as the number of bytes

n`k` as a multiple of 1024 bytes

n`m` as a multiple of 1048576 bytes

`–l` *lines*

`split` splits the input file into sections containing *lines* lines.

This corresponds to the old option *– lines*, which is still supported.

`–l` not specified:

`split` splits the input file into sections containing 1000 lines.

`––` End of the list of options. Must be specified if *file* begins with –.

file Name of the input file you want to split.

If you use a dash (–) as the name for *file*, `split` reads from standard input.

file not specified:

`split` reads from standard input.

name Name of the output files: The first output file is given the name *name*`aa`, the second receives the name *name*`ab`, and so on, up to *name*`zz`.

name must therefore be two characters shorter (or *number* characters if `–a` is specified) than the maximum permitted filename length (`NAME_MAX` bytes) in the relevant file system.

If you specify a value for *name*, the *file* argument is mandatory.

EXIT STATUS

0 The input file was split successfully

>0 An invalid option was specified or an error occurred

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The contents of the file example are to be split into several 20-line files:

```
$ split –l 20 example
$ ls
example
xaa
xab
xac
xad
```

Example 2

Every two lines from the standard input are to be written into files named out. Since the names of output files are explicitly specified (out) in this case, the minus sign (–) for standard input must not be omitted!

```
$ split –l 2 – out
What is true was always true
and will always remain true.
But what is not true, was never reality
and will never become reality.<CTRL-D>
$ ls
outaa
outab
```

SEE ALSO

bfs(1), csplit(1), statvfs(2).

User's Reference Manual

srchtxt(1)

NAME

`srchtxt` - display contents of, or search for a text string in,message databases

SYNOPSIS

`srchtxt` [*option ...*] [*text*]

DESCRIPTION

`srchtxt` is used to display the contents of AT&T-style message files or to search for text strings in such files.

The `srchtxt` utility can display all the message strings which are in a message database or search the message databases for a particular string [see also `mkmsgs(1)`]. Message databases are files in the directory `/usr/lib/locale/`*Locale*`/LC_MESSAGES`, where the directory name *Locale* stands for the name of the language in which the message strings are written. You can use the `–m` option to specify a different path name.

OPTIONS

No option specified:

The environment variable `LC_MESSAGES` determines the locale to be searched by `srchtxt`. If `LC_MESSAGES` is not set, the value of the environment variable `LANG` is used.

`–s` Suppresses printing of the sequence number of each message displayed.

`–l` *Locale*

The message database are located in the directory `/usr/lib/locale/`*Locale*`/LC_MESSAGES`.

`–l` not specified:

The value of the environment variable `LC_MESSAGES` determines the locale to be searched. If `LC_MESSAGES` is not set, the value of the environment variable `LANG` is used.

If `–l` is used in combination with `–m` *msgfile*, *Locale* is ignored for any *msgfile* which contains a slash, i.e. which represents a path name.

`–m` *msgfile*

`srchtxt` displays or searches *msgfile*. If more than one *msgfile* is named, the individual file names must be separated using commas.

If *msgfile* contains a slash, it is interpreted as a path name; otherwise, `srchtxt` will look for it in the directory determined as described above.

text Search for the specified *text* string and display each occurrence.

text can also take the form of a regular expression [see `ed(1)`].

text not specified:

All the text strings in the files accessed are displayed.

Output format

The displayed message texts are prefixed by their message sequence numbers and the file name, enclosed in angle brackets:

`<msgfile:msgnum>`

where *msgfile* is the file which contains the message text that follows, and *msgnum* is the sequence number of that message within *msgfile*.

The output is otherwise in the same format as described under `gettxt(1)` .

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

SEE ALSO

ed(1), exstr(1), gettxt(1), mkmsgs(1), gettxt(3C), setlocale(3C).

User's Reference Manual

ssh(1)

NAME

`ssh` - secure shell client (remote login program)

SYNOPSIS

`ssh` [`-l` *login_name*] `hostname` [*command*]

`ssh` [`-k`] [`-c` `blowfish|idea|des|3des|none`] [`-e` *ch*|`^`*ch*|`none`] [`-i` *identity_file*] [`-l` *login_name*] [`-n`] [`-o` *option*] [`-p` *port*] [`-q`] [`-t`] [`-v`] [`-x`] [`-X`] [`-C`] [`-L` *port*:*host*:*hostport*] [`-R` *port*:*host*:*hostport*] *hostname* [*command*]

DESCRIPTION

`ssh` (Secure Shell) is a program for logging into a remote machine and for executing commands in a remote machine. It is intended to replace `rlogin` and `rsh` and to provide secure encrypted communication between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

`ssh` connects and logs into the specified *hostname*. The user must prove his/her identity to the remote machine using one of several methods.

Firstly, if the machine the user logs in from is listed in `/etc/hosts.equiv` or `/etc/shosts.equiv` on the remote machine, and the user names are the same on both sides, the user is immediately permitted to log in. Secondly, if `.rhosts` or `.shosts` exists in the user's home directory on the remote machine and contains a line containing the name of the client machine and the name of the user on that machine, the user is permitted to log in. This form of authentication alone is normally not allowed by the server because it is not secure.

The second (and primary) authentication method is the `.rhosts` or `hosts.equiv` method combined with RSA (Rivest-Shamir-Adelman)-based host authentication. It means that if the login is permitted by `.rhosts`, `.shosts`, `/etc/hosts.equiv` or `/etc/shosts.equiv`, and additionally that it can verify the client's host key (see `$HOME/.ssh/known_hosts` and `/etc/ssh_known_hosts` in the **FILES** section), only then is login permitted. This authentication method closes security holes due to IP spoofing, DNS spoofing and routing spoofing. [Note to the administrator: `/etc/hosts.equiv`, `.rhosts` and the `rlogin/rsh` protocol in general are inherently insecure and should be disabled if security is desired.]

As a third authentication method, `ssh` supports RSA based authentication. The scheme is based on public-key cryptography: there are crypto systems where encryption and decryption are performed using separate keys and it is not possible to derive the decryption key from the encryption key. RSA is one such system. The idea is that each user creates a public/private key pair for authentication purposes. The server knows the public key and only the user knows the private key. The `$HOME/.ssh/authorized_keys` file lists the public keys that are permitted for logging in. When the user logs in, the `ssh` program tells the server which key pair it would like to use for authentication. The server checks if this key is permitted and if so sends the user (actually the `ssh` program running on behalf of the user) a challenge, a random number, encrypted by the user's public key. The challenge can only be decrypted using the proper private key. The user's client then decrypts the challenge using the private key, proving that he/she knows the private key but without disclosing it to the server.

`ssh` implements the RSA authentication protocol automatically. The user creates his/her RSA key pair by running `ssh-keygen(1)`. This stores the private key in `.ssh/identity` and the public key in `.ssh/identity.pub` in the user's home directory. The user should then copy the `identity.pub` to `.ssh/authorized_keys` in his/her home directory on the remote machine (the `authorized_keys` file corresponds to the conventional `.rhosts` file, and has one key per line, though the lines can be very long). After this, the user can log in without giving the password. RSA authentication is much more secure than `.rhosts` authentication.

The most convenient way to use RSA authentication may be with an authentication agent. See `ssh-agent(1)` for more information.

If other authentication methods fail, `ssh` prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

When the user's identity has been accepted by the server, the server either executes the given command or logs into the machine and gives the user a normal shell on the remote machine. All communication with the remote command or shell will be automatically encrypted.

If a pseudo-terminal has been allocated (normal login session), the user can disconnect with "~.", and suspend `ssh` with "~^Z". All forwarded connections can be listed with "~#", and if the session blocks waiting for forwarded X11 or TCP/IP connections to terminate, it can be backgrounded with "~&" (this should not be used while the user shell is active, as it can cause the shell to hang). All available escapes can be listed with "~?".

A single tilde character can be sent as "~~" (or by following the tilde with a character other than those described above). The escape character must always follow a newline to be interpreted as special. The escape character can be changed in configuration files or from the command line.

If no pseudo terminal has been allocated, the session is transparent and can be used to reliably transfer binary data. On most systems, setting the escape character to "none" will also make the session transparent even if a tty is used.

The session terminates when the command or shell on the remote machine exits and all X11 and TCP/IP connections have been closed. The exit status of the remote program is returned as the exit status of `ssh`.

If the user is using X11 (the `DISPLAY` environment variable is set), the connection to the X11 display is automatically forwarded to the remote side in such a way that any X11 programs started from the shell (or command) will go through the encrypted channel and the connection to the real X server will be made from the local machine. The user should not manually set `DISPLAY`. Forwarding of X11 connections can be configured from the command line or in configuration files.

The `DISPLAY` value set by `ssh` will point to the server machine, but with a display number greater than zero. This is normal and happens because `ssh` creates a "proxy" X server on the server machine for forwarding the connections over the encrypted channel.

`ssh` will also automatically set up X-authority data on the server machine. For this purpose, it will generate a random authorization cookie, store it in X-authority on the server and verify that any forwarded connections carry this cookie and replace it by the real cookie when the connection is opened. The real authentication cookie is never sent to the server machine (and no cookies are sent in plain text).

If the user is using an authentication agent, the connection to the agent is automatically forwarded to the remote side unless disabled from the command line or in a configuration file.

Forwarding of arbitrary TCP/IP connections over the secure channel can be specified either from the command line or in a configuration file. One possible application of TCP/IP forwarding is a secure connection to an electronic purse; another is going through firewalls.

`ssh` automatically maintains and checks a database containing RSA (Rivest-Shamir-Adelman) based identifications for all hosts it has ever been used with. The database is stored in `.ssh/known_hosts` in the user's home directory. Additionally, the file `/etc/ssh_known_hosts` is automatically checked for known hosts. Any new hosts are automatically added to the user's file. If a host's identification ever changes, `ssh` warns about this and disables password authentication to prevent a Trojan horse from getting the user's password. Another purpose of this mechanism is to prevent "man-in-the-middle" attacks (for details see: `http://www.ssh.com/tech/crypto/intro.html#timing-attack`) which could otherwise be used to circumvent the encryption. The `StrictHostKeyChecking` option (see below) can be used to prevent logins to machines whose host key is not known or has changed.

OPTIONS

`-c blowfish|idea|des|3des|none`

Selects the cipher to use for encrypting the session.

`blowfish`

Blowfish is an algorithm developed by Bruce Schneier. It is a block cipher with 64-bit block size

and variable length keys (up to 448 bits). It has gained a fair amount of acceptance in a number of applications. No attacks are known against it.

`idea` IDEA (International Data Encryption Algorithm) is an algorithm developed at ETH Zurich in Switzerland. It uses a 128 bit key and is generally considered to be very secure. It is currently one of the best known public algorithms. It is a fairly new algorithm, but it has already been around for several years. No practical attacks on it have been published despite numerous attempts to analyze it.

`des` DES is the data encryption standard, but is breakable by governments, large corporations and major criminal organizations.

`3des` 3DES (triple-des) is based on using `des` three times in an encrypt-decrypt-encrypt sequence with three different, unrelated keys. It is presumably more secure than `des`.

`3des` is used by default.

`none` Disables encryption entirely; it is only intended for debugging and it renders the connection insecure.

For more information see `http://www.ssh.com/tech/crypto/algorithms.html`.

`-e` *ch*|`^`*ch*|`none`
Sets the escape character for sessions with a pseudo-terminal to *ch* or `^`*ch* (default: ~). The escape character is only recognized at the beginning of a line. The escape character followed by a dot (.) closes the connection, followed by control-Z suspends the connection and followed by itself sends the escape character once. Setting the character to `none` disables any escapes and makes the session fully transparent.

`-f` Requests `ssh` to go to background after authentication. This is useful if `ssh` is going to prompt for passwords or passphrases, but the user wants it in the background. This implies `-n`. The recommended way to start X11 programs at a remote site is with something like "`ssh -f host xterm`".

`-i` *identity_file*
Selects the file from which the identity (private key) for RSA authentication is read. Default is `.ssh/identity` in the user's home directory. Identity files may also be specified on a per-host basis in the configuration file. It is possible to have multiple `-i` options (and multiple identities specified in configuration files).

`-k` Disables forwarding of Kerberos tickets/AFS tokens. This may also be specified on a per-host basis in the configuration file.

`-l` *login_name*
Specifies the user to log in as on the remote machine. This may also be specified on a per-host basis in the configuration file.

`-n` Redirects stdin from `/dev/null` (actually, prevents reading from stdin). This must be used when `ssh` is run in the background. A common trick is to use this to run X11 programs on a remote machine. For example, "`ssh -n shadows.cs.hut.fi emacs &`" will start an `emacs` on `shadows.cs.hut.fi` and the X11 connection will be automatically forwarded over an encrypted channel. The `ssh` program will be put in the background. (This does not work if `ssh` needs to prompt for a password or passphrase; see also the `-f` option.)

`-o` *option*
Can be used to give options in the format used in the configuration file. This is useful for specifying options for which there is no separate command-line flag. The option has the same format as a line in the configuration file.

`-p` *port*
Port to connect to on the remote host. This can be specified on a per-host basis in the configuration file.

`-q` Quiet mode. Causes all warning and diagnostic messages to be suppressed. Only fatal errors are

displayed.

`-t` Force pseudo-terminal allocation. This can be used to execute arbitrary screen-based programs on a remote machine, which can be very useful, for example, when implementing menu services.

`-v` Verbose mode. Causes `ssh` to print debugging messages about its progress. This is helpful in debugging connection, authentication and configuration problems.

`-x` Disables X11 forwarding. This can also be specified on a per-host basis in a configuration file.

`-X` Enables X11 forwarding.

`-C` Requests compression of all data (including stdin, stdout, stderr, and data for forwarded X11 and TCP/IP connections). The compression algorithm is the same used by `gzip` and the "level" can be controlled by the `CompressionLevel` option (see below). Compression is desirable on modem lines and other slow connections, but will only slow down things on fast networks. The default value can be set on a host-by-host basis in the configuration files; see the `Compress` option below.

`-L` *port*:*host*:*hostport*
Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to *port* on the local side. Then whenever a connection is made to this port, the connection is forwarded over the secure channel and a connection is made to *host*:*hostport* from the remote machine. Port forwardings can also be specified in the configuration file. Only root can forward privileged ports.

`-R` *port*:*host*:*hostport*
Specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side. This works by allocating a socket to listen to *port* on the remote side. Then whenever a connection is made to this port, the connection is forwarded over the secure channel and a connection is made to *host*:*hostport* from the local machine. Port forwardings can also be specified in the configuration file. Privileged ports can be forwarded only when logging in as root on the remote machine.

CONFIGURATION FILES

`ssh` obtains configuration data from the following sources (in this order): command line options, user's configuration file (`$HOME/.ssh/config`), and system-wide configuration file (`/etc/ssh_config`). For each parameter, the first obtained value will be used. The configuration files contain sections bracketed by "Host" specifications, and that section is only applied for hosts that match one of the patterns given in the specification. The matched host name is the one given at the command line.

Since the first obtained value for each parameter is used, more host-specific declarations should be given near the beginning of the file, with general defaults at the end.

The configuration file has the following format:

Empty lines and lines starting with "#" are comments.

Otherwise a line is of the format "keyword arguments". The possible keywords and their meanings are as follows (note that the configuration files are case-sensitive):

`Host` Restricts the following declarations (up to the next `Host` keyword) to be only for those hosts that match one of the patterns given after the keyword. "*" and "?" can be used as wildcards in the patterns. A single "*" as a pattern can be used to provide global defaults for all hosts. The host is the *hostname* argument given at the command line (i.e. the name is not converted to a canonical host name before matching).

`AFSTokenPassing`
Specifies whether to pass AFS tokens to remote hosts. The argument to this keyword must be "`yes`" or "`no`".

`BatchMode`
If set to "`yes`", passphrase/password querying will be disabled. This option is useful in scripts and other batch jobs where you have no user to supply the password. The argument must be "`yes`" or "`no`".

`Cipher`

Specifies the cipher to use for encrypting the session. Currently, `blowfish`, `idea`, `des`, `3des` and `none` are supported. The default is `3des`. Using `none` (no encryption) is intended only for debugging and will render the connection insecure (cf. `-c` option above).

`Compression`
Specifies whether to use compression. The argument must be "`yes`" or "`no`".

`CompressionLevel`
Specifies the compression level to use if compression is enabled. The argument must be an integer from 1 (fast) to 9 (slow, best). The default level is 6, which is good for most applications. The meaning of the values is the same as in GNU GZIP.

`ConnectionAttempts`
Specifies the number of tries (one per second) to make before falling back to `rsh` or exiting. The argument must be an integer. This may be useful in scripts if the connection sometimes fails.

`EscapeChar`
Sets the escape character (default: ~). The escape character can also be set at the command line. The argument should be a single character, "^" followed by a letter, or `none` to disable the escape character entirely (making the connection transparent for binary data).

`FallBackToRsh`
Specifies that if connecting via `ssh` fails due to a connection refused error (there is no `sshd` listening on the remote host), `rsh` should automatically be used instead (after a suitable warning about the session being unencrypted). The argument must be "`yes`" or "`no`".

`ForwardAgent`
Specifies whether the connection to the authentication agent (if any) will be forwarded to the remote machine. The argument must be "`yes`" or "`no`".

`ForwardX11`
Specifies whether X11 connections will be automatically redirected over the secure channel and `DISPLAY` set. The argument must be "`yes`" or "`no`".

`GlobalKnownHostsFile`
Specifies a file to use instead of `/etc/ssh_known_hosts`.

`HostName`
Specifies the real host name to log into. This can be used to specify nicknames or abbreviations for hosts. The default is the name given at the command line. Numeric IP addresses are also permitted (both at the command line and in `HostName` specifications).

`IdentityFile`
Specifies the file from which the user's RSA authentication identity is read (default `.ssh/identity` in the user's home directory). Additionally, any identities represented by the authentication agent will be used for authentication. The file name may use the tilde syntax to refer to a user's home directory. It is possible to have multiple identity files specified in configuration files; all these identities will be tried in sequence.

`KeepAlive`
Specifies whether the system should send "keepalive" messages to the other side. If they are sent, the death of the connection or crash of one of the machines will be properly noticed. However, this means that connections will die if the route is down temporarily and some people find this annoying.

The default is "`yes`" (to send keepalives), and the client will notice if the network goes down or the remote host dies. This is important in scripts and many users want it too.

To disable keepalives, the value should be set to "`no`" in both the server and the client configuration files.

`KerberosAuthentication`
Specifies whether Kerberos authentication will be used.

`KerberosTgtPassing`
Specifies whether a Kerberos TGT will be forwarded to the server. Note that TGT forwarding is normally not enabled in the server.

`LocalForward`
Specifies that a TCP/IP port on the local machine be forwarded over the secure channel to the given *host*:*port* on the remote machine. The first argument must be a port number and the second must be *host*:*port*. Multiple forwardings may be specified and additional forwardings can be given at the command line. Only root can forward privileged ports.

`PasswordAuthentication`
Specifies whether to use password authentication. The argument to this keyword must be "`yes`" or "`no`".

`Port` Specifies the port number to connect on the remote host. Default is 22.

`ProxyCommand`
Specifies the command to use to connect to the server. The command string extends to the end of the line and is executed with `/bin/sh`. In the command string, `%h` will be substituted by the host name to connect and `%p` by the port. The command can be basically anything and should read from its stdin and write to its stdout. It should eventually connect an `sshd` server running on some machine or execute "`sshd –i`" somewhere. Host key management will be performed using the `HostName` of the host being connected (defaulting to the name typed by the user).

Note that `ssh` can also be configured to support the `SOCKS` system using the `--with-socks` compile time configuration option.

`RemoteForward`
Specifies that a TCP/IP port on the remote machine be forwarded over the secure channel to the given *host*:*port* on the local machine. The first argument must be a port number and the second must be *host*:*port*. Multiple forwardings may be specified and additional forwardings can be given at the command line. Only root can forward privileged ports.

`RhostsAuthentication`
Specifies whether to try `.rhosts` based authentication. Note that this declaration only affects the client side and has no effect whatsoever on security. Disabling `.rhosts` authentication may reduce authentication time on slow connections when `.rhosts` authentication is not used. Most servers do not permit `RhostsAuthentication` because it is not secure (see `RhostsRSAAuthentication`). The argument to this keyword must be "`yes`" or "`no`".

`RhostsRSAAuthentication`
Specifies whether to try `.rhosts` based authentication with RSA host authentication. This is the primary authentication method for most sites. The argument must be "`yes`" or "`no`".

`RSAAuthentication`
Specifies whether to try RSA authentication. The argument to this keyword must be "`yes`" or "`no`". RSA authentication will only be attempted if the identity file exists or an authentication agent is running.

`StrictHostKeyChecking`
If this flag is set to "`yes`", `ssh` will never automatically add host keys to the `$HOME/.ssh/known_hosts` file and refuses to connect hosts whose host key has changed. This provides maximum protection against Trojan horse attacks. However, it can be somewhat annoying if you do not have good `/etc/ssh_known_hosts` files installed and frequently connect new hosts. Basically this option forces the user to manually add any new hosts. Normally this option is disabled and new hosts will automatically be added to the known host files. The host keys of known hosts will be verified automatically in either case. The argument must be "`yes`" or "`no`".

`User` Specifies the user to log in as. This can be useful if you have a different user name on different machines. This saves the trouble of having to remember to enter the user name at the command line.

`UserKnownHostsFile`
Specifies a file to use instead of `$HOME/.ssh/known_hosts`.

`UseRsh`
Specifies that `rlogin/rsh` should be used for this host. It is possible that the host does not support the `ssh` protocol. This causes `ssh` to immediately execute `rsh`. All other options (except `HostName`) are ignored if this has been specified. The argument must be "`yes`" or "`no`".

ENVIRONMENT

ssh will normally set the following environment variables:

DISPLAY
: The DISPLAY variable indicates the location of the X11 server. It is automatically set by ssh to point to a value of the form "*hostname*:*n*" where *hostname* indicates the host where the shell runs and *n* is an integer ≥ 1. ssh uses this special value to forward X11 connections over the secure channel. The user should normally not set DISPLAY explicitly, as that will render the X11 connection insecure (and will require the user to manually copy any required authorization cookies).

HOME
: Set to the path of the user's home directory.

LOGNAME
: Synonym for USER; set for compatibility with systems that use this variable.

MAIL
: Set to point the user's mailbox.

PATH
: Set to the default PATH, as specified when compiling ssh or /etc/default/login.

SSH_AUTHENTICATION_FD
: This is set to an integer value if you are using the authentication agent and a connection to it has been forwarded. The value indicates a file descriptor number used for communicating with the agent. On some systems, SSH_AUTHENTICATION_SOCKET may be used instead to indicate the path of a Unix domain socket used to communicate with the agent (this method is less secure and is only used on systems that do not support the first method).

SSH_CLIENT
: Identifies the client end of the connection. The variable contains three space-separated values: client ip address, client port number and server port number.

SSH_TTY
: This is set to the name of the tty (path to the device) associated with the current shell or command. If the current session has no tty, this variable is not set.

TZ
: The timezone variable is set to indicate the present timezone if it was set when the daemon was started (i.e. the daemon passes the value on to new connections).

USER
: Set to the name of the user logging in.

Additionally, ssh reads /etc/default/login and $HOME/.ssh/environment and adds lines of the format VARNAME=*value* to the environment.

FILES

$HOME/.ssh/known_hosts
: Records host keys for all hosts the user has logged into (that are not in /etc/ssh_known_hosts). See sshd(8) .

$HOME/.ssh/random_seed
: Used for seeding the random number generator. This file contains sensitive data and should be read/write for the user and not accessible for others. This file is created the first time the program is run and updated automatically. The user should never need to read or modify this file.

$HOME/.ssh/identity
: Contains the RSA authentication identity of the user. This file contains sensitive data and should be readable by the user but not accessible by others. It is possible to specify a passphrase when generating the key; the passphrase will be used to encrypt the sensitive part of this file using IDEA (cf. -c option above).

$HOME/.ssh/identity.pub
: Contains the public key for authentication (public part of the identity file in human-readable form). The contents of this file should be added to $HOME/.ssh/authorized_keys on all machines where you wish to log in using RSA authentication. This file is not sensitive and can (but need not) be readable by anyone. This file is never used automatically and is not necessary; it is only provided for the convenience of the user.

`$HOME/.ssh/config`
: This is the per-user configuration file. The format of this file is described above. This file is used by the `ssh` client. This file does not usually contain any sensitive information, but the recommended permissions are read/write for the user and not accessible by others.

`$HOME/.ssh/authorized_keys`
: Lists the RSA keys that can be used for logging in as this user. The format of this file is described in the `sshd(8)` manual page. In its simplest form, the format is the same as the `.pub` identity files (that is, each line contains the number of bits in modulus, public exponent, modulus and comment fields, separated by spaces). This file is not highly sensitive, but the recommended permissions are read/write for the user and not accessible by others.

`/etc/ssh_known_hosts`
: System-wide list of known host keys. This file should be prepared by the system administrator to contain the public host keys of all machines in the organization. This file should be world-readable. This file contains public keys, one per line, in the following format (fields separated by spaces): system name, number of bits in modulus, public exponent, modulus and optional comment field. When different names are used for the same machine, all such names should be listed, separated by commas. The format is described on the `sshd(8)` manual page.

  The canonical system name (as returned by name servers) is used by `sshd` to verify the client host when logging in; other names are needed because `ssh` does not convert the user-supplied name to a canonical name before checking the key, because someone with access to the name servers would then be able to fool host authentication.

`/etc/ssh_config`
: System-wide configuration file. This file provides defaults for those values that are not specified in the user's configuration file and for those users who do not have a configuration file. This file must be world-readable.

`$HOME/.rhosts`
: This file is used in `.rhosts` authentication to list the host/user pairs that are permitted to log in. (Note that this file is also used by `rlogin` and `rsh`, which makes using this file insecure.) Each line in the file contains a host name (in the canonical form returned by name servers) and then a user name on that host, separated by a space. On some machines, this file may need to be world-readable if the user's home directory is on a NFS partition, because `sshd` reads it as root. Additionally, this file must be owned by the user and must not have write permissions for anyone else. The recommended permission for most machines is read/write for the user and not accessible by others.

  Note that by default `sshd` will be installed so that it requires successful RSA host authentication before permitting `.rhosts` authentication. If your server machine does not have the client's host key in `/etc/ssh_known_hosts`, you can store it in `$HOME/.ssh/known_hosts`. The easiest way to do this is to connect back to the client from the server machine using `ssh`; this will automatically add the host key in `$HOME/.ssh/known_hosts`.

`$HOME/.shosts`
: This file is used in exactly the same way as `.rhosts`. The purpose of having this file is to be able to use `.rhosts` authentication with `ssh` without permitting login with `rlogin` or `rsh`.

`/etc/hosts.equiv`
: This file is used during `.rhosts` authentication. It contains canonical hosts names, one per line (the full format is described on the `sshd(8)` manual page). If the client host is found in this file, login is automatically permitted provided client and server user names are the same. Additionally, successful RSA host authentication is normally required. This file should only be writeable by root.

`/etc/shosts.equiv`
: This file is processed exactly as `/etc/hosts.equiv`. This file may be useful to permit logins using `ssh` but not using `rsh/rlogin`.

`/etc/sshrc`
: Commands in this file are executed by `ssh` when the user logs in, just before the user's shell (or command) is started. See the `sshd(8)` manual page for more information.

`$HOME/.ssh/rc`
: Commands in this file are executed by `ssh` when the user logs in. just before the user's shell (or command) is started. See the `sshd(8)` manual page for more information.

INSTALLATION

`ssh` is normally installed as suid root. It needs root privileges only for `.rhosts` authentication (`.rhosts` authentication requires that the connection must come from a privileged port and allocating such a port requires root privileges). It also needs to be able to read `/etc/ssh_host_key` to perform RSA host authentication. It is possible to use `ssh` without root privileges, but `.rhosts` authentication will then be disabled. `ssh` drops any extra privileges immediately after the connection to the remote host has been made.

REFERENCES

`http://www.ssh.com`

`http://www.ssh.org`

SEE ALSO

`rlogin(1)`, `rsh(1)`, `scp(1)`, `ssh-add(1)`, `ssh-agent(1)`, `ssh-keygen(1)`, `telnet(1)`, `sshd(8)`
.

User's Reference Manual

ssh-add(1)

NAME

`ssh-add` - adds identities for the authentication agent

SYNOPSIS

`ssh-add` [`-l`] [`-d`] [`-D`] [*file ...*]

DESCRIPTION

`ssh-add` adds identities to the authentication agent, `ssh-agent(1)`. When run without arguments, it adds the file `$HOME/.ssh/identity`. Alternative file names can be given at the command line. If any file requires a passphrase, `ssh-add` prompts the user for the passphrase. If the user is using X11, the passphrase is requested using a small X11 program; otherwise it is read from the user's tty. (Note: it may be necessary to redirect stdin from `/dev/null` to get the passphrase requested using X11.)

The authentication agent must be running and must be an ancestor of the current process for `ssh-add` to work.

OPTIONS

`-l` Lists all identities currently represented by the agent.

`-d` Instead of adding the identity, removes the identity from the agent.

`-D` Deletes all identities from the agent.

FILES

`$HOME/.ssh/identity`

Contains the RSA authentication identity of the user. This file should not be readable by anyone but the user. It is possible to specify a passphrase when generating the key; this passphrase is then used to encrypt the private part of the file. This is the default file added by `ssh-add` when no other files have been specified.

If `ssh-add` needs a passphrase, it will read the passphrase from the current terminal if it was run from a terminal. If `ssh-add` does not have a terminal associated with it but `DISPLAY` is set, it will open an X11 window to read the passphrase. This is particularly useful when calling `ssh-add` from a `.Xsession` file or related script. (Note that on some machines it may be necessary to redirect the input from `/dev/null` to make this work.)

SEE ALSO

`ssh(1)`, `ssh-agent(1)`, `ssh-keygen(1)`, `sshd(8)`.

User's Reference Manual

ssh-agent(1)

NAME

`ssh-agent` - authentication agent

SYNOPSIS

`ssh-agent` *command*

DESCRIPTION

`ssh-agent` is a program that stores authentication private keys. The idea is that `ssh-agent` is started at the beginning of an X-session or a login session and all other windows or programs are started as children of the `ssh-agent` program (the *command* normally starts X or is the user shell). Programs started under the agent inherit a connection to the agent and the agent is automatically used for RSA (Rivest-Shamir-Adelman) authentication when logging in to other machines using `ssh(1)`.

The agent initially does not have any private keys. Keys are added using `ssh-add(1)`. When executed without arguments, `ssh-add` adds the `$HOME/.ssh/identity` file. If the identity has a passphrase, `ssh-add` prompts for the passphrase (using a small X11 application if running under X11, or from the terminal if running without X). It then sends the identity to the agent. Several identities can be stored in the agent; the agent can automatically use any of these identities. `ssh-add –l` displays the identities currently held by the agent.

The idea is that the agent is run on the user's local PC, laptop or terminal. Authentication data need not be stored on any other machine; authentication passphrases never go over the network. However, the connection to the agent is forwarded over `ssh` remote logins and the user can thus use the privileges given by the identities anywhere in the network in a secure way.

A connection to the agent is inherited by child programs. There are two alternative methods for inheriting the agent. The preferred method is to have an open file descriptor which is inherited and have an environment variable (`SSH_AUTHENTICATION_FD`) contain the number of this descriptor. This restricts access to the authentication agent to only those programs that are siblings of the agent. It is fairly difficult in this case even for root to get unauthorized access to the agent.

An alternative method is used on some machines. A Unix domain socket is created (`/tmp/ssh_agent.*`) and the name of this socket is stored in the `SSH_AUTHENTICATION_SOCKET` environment variable. The socket is made accessible to the current user only. This method is easily abused by root or another instance of the same user. The socket is only used if ssh is unable to find a file descriptor that would not be closed by shells.

The agent exits automatically when the command given at the command line terminates.

FILES

`$HOME/.ssh/identity`
: Contains the RSA authentication identity of the user. This file should not be readable by anyone but the user. It is possible to specify a passphrase when generating the key; this passphrase is then used to encrypt the private part of the file. This file is not used by `ssh-agent`, but is normally added to the agent using `ssh-add` at login time.

`/tmp/ssh_agent.`*pid*
: Unix domain sockets used to contain the connection to the authentication agent. These sockets should only be readable by the owner. The sockets should be removed automatically when the agent exits.

SEE ALSO

`ssh(1)`, `ssh-add(1)`, `ssh-keygen(1)`, `sshd(8)`.

User's Reference Manual

ssh-keygen(1)

NAME

`ssh-keygen` - authentication key generation

SYNOPSIS

`ssh-keygen` [`–b` *bits*] [`–N` *new_passphrase*] [`–C` *comment*]

`ssh-keygen –p` [`–P` *old_passphrase*] [`–N` *new_passphrase*]

`ssh-keygen –c` [`–P` *passphrase*] [`–C` *comment*]

DESCRIPTION

`ssh-keygen` generates and manages authentication keys for `ssh(1)` . Normally each user wishing to use `ssh` with RSA (Rivest-Shamir-Adelman) authentication runs this once to create the authentication key in `$HOME/.ssh/identity`. Additionally, the system administrator may use this to generate host keys.

Normally this program generates the key and asks for a file in which to store the private key. The public key is stored in a file with the same name but with "`.pub`" appended. The program also prompts for a passphrase. The passphrase may be empty to indicate no passphrase (host keys must have an empty passphrase) or it may be a string of arbitrary length. Good passphrases are 10-30 characters long and are not simple sentences or otherwise easily guessed (English prose has only 1-2 bits of entropy per word and provides very bad passphrases). The passphrase can be changed later using the `–p` option.

There is no way to recover a lost passphrase. If the passphrase is lost or forgotten, you will have to generate a new key and copy the corresponding public key to other machines.

There is also a comment field in the key file, which is simply provided for the user's convenience to help identify the key. The comment can tell what the key is for or whatever is useful. The comment is initialized to *user@host* when the key is created, but can be changed using the `–c` option.

OPTIONS

`–b` *bits*
: Specifies the number of bits in the key being created. Minimum is 512 bits. Generally 1024 bits is considered sufficient; higher key sizes no longer improve security but make things slower. The default is 1024 bits.

`–c`
: Requests changing the comment in the private and public key files. The program will prompt for the file containing the private keys, for passphrase if the key has one and for the new comment.

`–p`
: Requests changing the passphrase of a private key file instead of creating a new private key. The program will prompt for the file containing the private key, for the old passphrase and twice for the new passphrase.

`–C` *comment*
: Provides the new comment.

`–N` *new_passphrase*
: Provides the new passphrase.

`–P` *old_passphrase*
: Provides the (old) passphrase.

FILES

`$HOME/.ssh/random_seed`
: Used for seeding the random number generator. This file should not be readable by anyone but the user. This file is created the first time when the program is run and is updated every time.

`$HOME/.ssh/identity`
: Contains the RSA authentication identity of the user. This file should not be readable by anyone but the

user. It is possible to specify a passphrase when generating the key; this passphrase is then used to encrypt the private part of the file using `IDEA` [see `ssh(1)`]. This file is not automatically accessed by `ssh-keygen`, but is offered as the default file for the private key.

`$HOME/.ssh/identity.pub`
Contains the public key for authentication. The contents of this file should be added to `$HOME/.ssh/authorized_keys` on all machines where you wish to log in using RSA authentication. There is no need to keep the contents of this file secret.

SEE ALSO

`ssh(1)`, `ssh-add(1)`, `ssh-agent(1)`, `sshd(8)`.

User's Reference Manual

strchg(1)

NAME

`strchg -` change stream configuration

SYNOPSIS

Format 1

`strchg –h` *module1*[,*module2* ...]

Format 2

`strchg –p` [*option*]

Format 3

`strchg –f` *file*

DESCRIPTION

`strchg` is used to alter the configuration of the stream associated with the user's standard input. `strchg` pushes modules on and/or pops modules off the stream. Only the system administrator or the owner of a device may alter the configuration of the associated stream. The command will fail if run by any other user or if the the user does not have read permission on the stream and does not have system administrator privileges.

If you push the modules for a stream in the wrong order, you may find that the stream does not function as you expect. For example, if the line discipline module is not pushed in the correct place for a terminal, you may find that your terminal no longer responds to any input.

Format 1: Pushing modules onto a stream

`strchg –h` *module1*[,*module2* ...]

You can specify the names of one or more pushable streams modules as arguments. These modules are pushed (activated) in the given order, i.e. *module1* is pushed first, *module2* is pushed second, etc. The module names must be separated by commas. There must be no spaces within the module list.

Format 2: Popping modules off the stream

`strchg –p` [*option*]

If no option is specified with `–p`, `strchg` pops the topmost module from the stream (deactivates it).

`–a` If you specify `–a` together with `–p`, all the modules above the topmost driver are popped. The `–a` option may only be specified together with `–p`, and not in combination with `–u`.

`–u` *module*
: All modules above but not including *module* are popped off the stream. The `–u` option may only be specified together with `–p`, and not in combination with `–a`.

Format 3: Defining a stream configuration

`strchg –f` *file*

file contains a list of modules representing the required configuration of the stream. Each module name must appear on a separate line, with the first line containing the name of the topmost module and the last line the name of the lowest one, i.e. the module that should be closest to the driver. `strchg` determines the current configuration of the stream and pops and pushes the necessary modules in order until the chosen configuration is achieved.

EXAMPLES

Example 1

The following command pushes the module `ldterm` on the stream associated with your standard input:

```
$ strchg –h ldterm
```

Example 2

The following command pops the topmost module from the stream associated with `/dev/term/24`. You must be the owner of this device or have system administrator privileges to run this command:

```
$ strchg –p </dev/term/24
```

Example 3

If the file `conf` contains the following three lines:

```
compat
ldterm
ptem
```

and you run the command:

```
$ strchg -f conf
```

`strchg` will configure your standard input stream with the module `ptem` pushed directly over the driver, followed by the module `ldterm` above it, and with the module `compat` as the topmost module closest to the stream head.

SEE ALSO

`strconf`(1), `streamio`(7).

User's Reference Manual

strconf(1)

NAME

`strconf -` query stream configuration

SYNOPSIS

`strconf [ –t| –m` *module* `]`

DESCRIPTION

`strconf` is used to query the configuration of the stream associated with the user's standard input.

OPTIONS

No option specified:

`strconf` lists all the modules in the stream as well as the topmost driver. The list has one name per line; the first item is the topmost module on the stream (if one exists), and the last item is the name of the driver.

`–t` Only the topmost module (if one exists) is listed.

`–m` *module*

`strconf` checks if the named module is present on a stream. If it is, `strconf` prints the message `yes` and returns an exit status of zero. If not, it prints the message `no` and returns a non-zero exit status.

The `–t` and `–m` options are mutually exclusive.

EXIT STATUS

`0` The check on the stream was successful. With the `–m` or `–t` option, a zero exit status means that the named or topmost module is present.

not `0`

The `–m` or `–t` option was used and the specified module is not present. A non-zero exit status is also returned for various error conditions such as the use of invalid options or failure of an `ioctl` on the stream.

EXAMPLES

Example 1

The `strconf` command without arguments produces the following output for a stream that has only the module `ldterm` pushed above the `ports` driver:

```
ldterm
ports
```

Example 2

The following command checks if the module `ldterm` is on the stream:

```
$ strconf –m ldterm
```

If the module is present, `strconf` prints the message:

```
yes
```

and returns an exit status of zero.

SEE ALSO

`strchg(1)`, `streamio(7)`.

User's Reference Manual

strings(1)

NAME

`strings -` find printable strings in object or binary files

SYNOPSIS

`strings` [`–a`] [`–o`|`–t` *format*] [`–n` *number*] [`––`] *file* ...

DESCRIPTION

`strings` looks for strings in binary files and writes them on standard output. A string is by default any sequence of 4 or more printable ASCII characters ending with a newline or a null byte [see `ascii(5)`]. `strings` is useful for identifying random object files and many other things.

OPTIONS

`–a` `strings` searches the entire file for printable strings.

This corresponds to the old option `–`, which is still supported.

`–a` not specified:

`strings` only looks in the initialized data space of object files.

`–o` `strings` precedes each string by its offset in the file.

`–t` *format*

Each string is output preceded by its offset in the file. The *format* of the positioning specification is defined as follows:

- `d` Decimal positioning specification
- `o` Octal positioning specification
- `x` Hexadecimal positioning specification

`–n` *number*

Defines a strings as any sequence of `number` or more printable characters ending with a newline or a null byte.

This corresponds to the old option *–number*, which is still supported.

`–n` not specified:

A string is any sequence of 4 or more printable characters ending with a newline character or a null byte.

`––` End of the list of options. Must be specified if *file* begins with `–`.

file Name of the file in which `strings` is to look for printable strings.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Finding all printable strings in the executable binary file `a.out`:

```
$ strings a.out
Hello, user
```

This could be the output for a file that originally held the following source code:

```
#include <stdio.h>
main()
{
printf("Hello, user\n");
}
```

SEE ALSO

nm(1), od(1), ascii(5).

User's Reference Manual

stty(1)

NAME

`stty -` check and change terminal arguments (set terminal type)

SYNOPSIS

`stty` [{–a | –g}]

`stty` [*argument ...*]

DESCRIPTION

`stty` can be used to check or change terminal arguments.

Note:

Depending on the implementation, some `stty` arguments such as `parext`, `iexten` and `stflush` are accepted but not interpreted by the driver, so these arguments will not have any effect.

OPTIONS

No option and no argument specified:

`stty` reports some of the argument settings for the device that is its current standard input.

`–a` Reports all currently valid terminal argument settings.

`–g` Reports current settings in a form that can be used as input to another `stty` command.

Arguments for setting input/output characteristics

The following section describes the various arguments available when setting I/O characteristics for the terminal that is `stty`'s current standard input. The I/O arguments are listed in five main groups:

- Data transfer control arguments
- Input control arguments
- Output control arguments
- Local features
- Control assignments.

An additional group constitutes combinations of arguments from these five groups.

Explanations related to parenthesized arguments are also indicated within parentheses.

Note that even though many combinations of arguments make no sense, `stty` performs no sanity checking.

Data transfer control arguments

The following table provides an overview of the various arguments that can be used to control data transfer:

Argument	Brief description
`parenb`	Enable parity generation and detection
`parext`	Extended parity generation and detection
`parodd`	Odd parity
`csize`	Define character size: `cs5`, `cs6`, `cs7` or `cs8`
`0`	Hang up lines

	immediately
`number`	Set input baud rate to *number*
`ispeed` *number*	Set terminal input baud rate to *number*
`ospeed` *number*	Set terminal output baud rate to *number*
`hupcl`	Hang up connection on last `close`(2)
`hup`	Same as `hupcl`
`cstopb`	Use two stop bits per character
`cread`	Enable receiver
`clocal`	Modem control signals not supported
`loblkl`	Block output from a non-current layer

`parenb` (`–parenb`)
Enable (disable) parity generation and detection.

`parext` (`–parext`)
Enable (disable) extended parity generation and detection for mark and space parity.

`parodd` (`–parodd`)
Select odd (even) parity, or mark (space) parity if `parext` is enabled. These arguments are only effective in combination with the `parenb` and `parext` arguments.

csize
Define character size. *csize* can have any of the following values: `cs5`, `cs6`, `cs7`, or `cs8`. The number specifies how many bits are used to represent a character during I/O (excluding the parity bit).

`0` Hang up lines immediately. This applies to all terminal lines – not just modem lines. A `SIGHUP` signal is sent to all processes attached to the terminal.

number
Set baud rate to the *number* given, if possible. Note that not all baud rates are supported by all hardware interfaces [see `termio`(7)].

`ispeed` *number*
Set terminal input baud rate to the *number* given. Note that not all devices support different input and output baud rates. If the input baud rate is set to zero, the input baud rate will be set to the value of the output baud rate.

Possible values for *number* are: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400.

`ospeed` *number*
Set terminal output baud rate to the *number* given. Note that not all devices support different input and output baud rates. If the output baud rate is set to zero, the line will be hung up immediately.

Possible values for *number* are: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400.

`hupcl` (`–hupcl`)
Hang up (do not hang up) modem connection after last call to `close`(2) .

`hup` (`–hup`)

Same as `hupcl` (`–hupcl`).

`cstopb` (`–cstopb`)
: Use two (one) stop-bit(s) per character.

`cread` (`–cread`)
: Enable (disable) the receiver. If `cread` is not set, no characters can be received.

`clocal` (`–clocal`)
: Assume a local direct connection without (with) modem control. Modem control signals are therefore not supported (or supported).

`loblk` (`–loblk`)
: Block (do not block) output from a non-current layer.

Input control arguments

The following arguments can be used to control the input of data:

Argument	Brief description
`ignbrk`	Ignore break on input
`brkint`	`SIGINT` on break
`ignpar`	Ignore parity errors
`parmrk`	Mark characters with parity errors
`inpck`	Enable input parity checking
`istrip`	Mask input characters to 7 bits
`inlcr`	Map newline to carriage return on input
`igncr`	Ignore carriage return on input
`icrnl`	Map carriage return to newline on input
`iuclc`	Map uppercase letters to lowercase on input
`ixon`	Enable START/STOP output control
`ixany`	Allow any character to restart output
`ixoff`	Do not send STOP character when input queue is nearly full
`imaxbel`	Echo ASCII BEL character when input line is too long

`ignbrk` (`–ignbrk`)
: Ignore (do not ignore) break on input.

`brkint` (`–brkint`)
: Send (do not send) `SIGINT` on break.

`ignpar` (`–ignpar`)
: Ignore (do not ignore) parity errors.

`parmrk` (`–parmrk`)
: Mark (do not mark) characters with parity errors.

`inpck` (`–inpck`)
: Enable (disable) parity checking on input.

`istrip` (`–istrip`)
: Mask (do not mask) input characters to 7 bits.

`inlcr` (`–inlcr`)
: Map (do not map) newline character to carriage return on input.

`igncr` (`–igncr`)
: Ignore (do not ignore) carriage returns on input.

`icrnl` (`–icrnl`)
: Map (do not map) carriage return to newline character on input.

`iuclc` (`–iuclc`)
: Map (do not map) uppercase letters to lowercase on input.

`ixon` (`–ixon`)
: Enable (disable) START/STOP output control. Output is stopped by sending an ASCII DC3 and started by sending an ASCII DC1 [see `ascii(5)`].

`ixany` (`–ixany`)
: Allow any character (only the ASCII character DC1) to restart output.

`ixoff` (`–ixoff`)
: Request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

`imaxbel` (`–imaxbel`)
: Echo (do not echo) the ASCII BEL character when the input line is too long.

Output control arguments

The following arguments can be used to control the output of data:

Argument	Brief description
`opost`	Post-process output in accordance with subsequent arguments
`oluc`	Map lowercase letters to uppercase on output
`onlcr`	Map newline to carriage return on output
`ocrnl`	Map carriage return to newline on output
`onocr`	Do not output carriage returns in column 0

onlret	Newline also performs the carriage return function
ofill	Use fill characters for data transmission delays
ofdel	Use abort character DEL 0x7 as fill character
cr0,1,2,3	Select extent of delay for carriage returns on output
nl0 nl1	Select extent of delay for newline characters on output
tab0,1,2,3	Select extent of delay for horizontal tabs on output
bs0,1	Select extent of delay for backspaces on output
ff0,1	Select extent of delay for form feeds on output
vt0,1	Select extent of delay for vertical tabs on output

opost (–opost)
: Post-process output in accordance with the other specified arguments (do not post-process output: ignore all other output control arguments).

olcuc (–olcuc)
: Map (do not map) lowercase letters to uppercase on output.

onlcr (–onlcr)
: Map (do not map) newline character to carriage return on output.

ocrnl (–ocrnl)
: Map (do not map) carriage return to newline character on output.

onocr (–onocr)
: Do not (do) output carriage return in column 0.

onlret (–onlret)
: A newline character also performs (does not perform) the carriage return function on output.

ofill (–ofill)
: Use fill characters (use timing) for data transmission delays.

ofdel (–ofdel)
: Use abort character DEL 0x7f (null character NUL 0x0) as fill character.

cr0 cr1 cr2 cr3
: Select extent of delay for carriage returns on output [see termio(7)].

nl0 nl1

Select extent of delay for newline characters on output [see termio(7)].

`tab0 tab1 tab2 tab3`
Select extent of delay for horizontal tabs on output [see termio(7)].

`bs0 bs1`
Select extent of delay for backspaces on output [see termio(7)].

`ff0 ff1`
Select extent of delay for form feeds on output [see termio(7)].

`vt0 vt1`
Select extent of delay for vertical tabs on output [see termio(7)].

Local features

Argument	Brief description
`isig`	Enable checking of input characters against control characters `intr`, `quit`, and `swtch`
`icanon`	Enable canonical input
`xcase`	Enable canonical upper/lowercase presentation
`echo`	Echo back every character typed
`echoe`	Echo erase character as backspace-space-backspace string
`echok`	Output an additional newline after the kill character
`lfck`	Same as `echok`
`echonl`	Output newline characters
`noflsh`	Disable flushing of the I/O queue
`stwrap`	Disable truncation of lines on a synchronous line
`tostop`	Send `SIGTTOU` when background processes write to the terminal
`echoctl`	Output control characters in the form: ^*char*
`echoprt`	Output erase character when a character is "erased"
`echoke`	Output a backspace-space-backspace sequence to erase the entire line to be killed
`flusho`	Flush output queue
`pendin`	Retype pending input
`iexten`	Enable extended functions for input data
`stflush`	Flush output queue after every `write`
`stappl`	Use application mode on a synchronous line

`isig` (`–isig`)
Enable (disable) the checking of each input character against the special control characters `intr`, `quit`, and `swtch` (see **Control Assignments**). If a match is found, the function associated with the control character is executed (not executed).

`icanon` (`–icanon`)
Enable (disable) canonical input, i.e. the special functions of the characters erase and kill are (are not) performed on input [see termio(7)].

`xcase` (`–xcase`)
Enable (disable) canonical uppercase/lowercase presentation.

`echo` (`–echo`)
Echo back (do not echo back) every character typed.

`echoe` (`–echoe`)

Echo (do not echo) erase characters as a backspace-space-backspace string.

Caution:

Setting the `echoe` argument will erase the character overwritten with erase on many terminals; however, since this mode does not keep track of the column position, problems may arise with escaped characters, tabs, and backspaces.

`echok` (`–echok`)
Output (do not output) an additional newline after a kill character.

`lfkc` (`–lfkc`)
The same as `echok` (`–echok`); obsolete.

`echonl` (`–echonl`)
Output (do not output) newline characters.

`noflsh` (`–noflsh`)
Disable (enable) the flushing of input and output queues on detection of an `intr`, `quit`, or `swtch` signal.

`stwrap` (`–stwrap`)
Disable (enable) truncation of lines longer than 79 characters on a synchronous line.

`tostop` (`–tostop`)
Send (do not send) `SIGTTOU` when background processes write to the terminal.

`echoctl` (`–echoctl`)
Output (do not output) control characters as `^`*char* (delete as `^?`).

`echoprt` (`–echoprt`)
Output (do not output) erase character when a character is "erased".

`echoke` (`–echoke`)
Output a backspace-space-backspace sequence to erase entire line to be killed.

`flusho` (`–flusho`)
Flush (do not flush) the output queue.

`pendin` (`–pendin`)
Retype (do not retype) pending input at next read or input character.

`iexten` (`–iexten`)
Enable (disable) extended functions for input data. These include the special characters `werase`, `reprint`, `discard` and `lnext`, and the local features `tostop`, `echoctl`, `echoprt`, `echoke`, `flusho` and `pendin`.

`stflush` (`–stflush`)
Flush (do not flush) output queue on a synchronous line after every `write`.

`stappl` (`–stappl`)
Use application mode (use line mode) on a synchronous line.

Control assignments

control-character c
Set *control-character* to *c*, where *control-character* can be:

`ctab`, `discard`, `dsusp`, `eof`, `eol`, `eol2`, `erase`,`intr`, `kill`, `lnext`, `quit`, `reprint`, `start`, `stop`, `susp`, `swtch`, `werase`, `min` or `time` (`min` and `time` are used with `–icanon`; `ctab` is used with `–stappl`). If *c* is preceded by the ^ character, then the value used is the corresponding CTRL character (e.g. ^D corresponds to <CTRL-D>). ^? is interpreted as <DEL>, and ^- is interpreted as undefined.

`min` *number*
`time` *number*
Set the value of `min` or `time` to *number*. `min` and `time` are used with `–icanon` [see `termio(7)`].

`line` *i*

Set line discipline to *i* (0 < *i* < 127).

Combination modes

The following combination modes may be used to control the terminal:

Argument	Brief description
`evenp`	Enable `parenb` and `cs7`
`parity`	Same as `evenp`
`oddp`	Enable `parenb`, `cs7` and `parodd`
`spacep`	Enable `parenb`, `cs7` and `parext`
`markp`	Enable `parenb`, `cs7`, `parodd` and `parext`
`-evenp`	Disable `parenb`, and enable `cs8`
`-parity`	Same as `parity`
`-oddp`	Disable `parenb and parodd`, and enable `cs8`
`-spacep`	Disable `parenb` and `parext`, and enable `cs8`
`-markp`	Disable `parenb`, `parodd`, and `parext`, and enable `cs8`
`raw`	Enable raw mode
`nl`	Disable `icrnl`, `ocrnl`, `inlcr`, `igncr`, `ocrnl`, and `onlret`
`lcase`	Enable `xcase`, `iuclc`, and `oluc`
`LCASE`	Same as `lcase`
`tabs`	Preserve tabs when printing
`ek`	Reset erase and kill to original values
`sane`	Reset all terminal arguments to reasonable values
`term`	Set all values for a terminal type *term*
`async`	Set normal asynchronous communication

`evenp` or `parity`
: Enable `parenb` and `cs7`.

`oddp` Enable `parenb`, `cs7`, and `parodd`.

`spacep`
: Enable `parenb`, `cs7`, and `parext`.

`markp`
: Enable `parenb`, `cs7`, `parodd`, and `parext`.

`–parity` or `–evenp`
: Disable `parenb`, and enable `cs8`.

`–oddp`
: Disable `parenb` and `parodd`, and enable `cs8`.

`–spacep`
: Disable `parenb` and `parext`, and enable `cs8`.

`–markp`
: Disable `parenb`, `parodd`, and `parext`, and enable `cs8`.

`raw` (`–raw` or `cooked`)
: Enable (disable) raw input and output; raw mode is equivalent to: `cs8`, with no `erase`, `kill`, `intr`, `quit`, `swtch`, `eof`, or output post-processing, and no parity bit.

`nl` (`–nl`)
: Disable (enable) `icrnl`, and `onlcr`. In addition, `–nl` disables `inlcr`, `igncr`, `ocrnl`, and `onlret`.

`lcase` (`–lcase`)
: Enable (disable) `xcase`, `iuclc`, and `olcuc`.

`LCASE` (`–LCASE`)
: Same as `lcase` (`–lcase`).

`tabs` (`–tabs` or `tab8`)
: Preserve tabs (expand to spaces) when printing.

`ek`
: Reset control characters erase and kill back to system-specific default values.

`sane`
: Reset all terminal arguments to some reasonable values.

term
: Set all arguments suitable for the terminal type *term*, where *term* is one of the following: tty33, tty37, vt05, tn300, ti700, or tek.

`async`
: Set normal asynchronous communication where clock settings are `xcibrg`, `rcibrg`, `tsetcoff` and `rsetcoff` [see also `termiox(7)`].

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Check current settings:

```
$ stty –a
speed 38400 baud;
intr = DEL; quit = ^|; erase = ^h;
kill = @; eof = ^d; eol <undef>; eol2 = <undef>; swtch <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = <undef>;
rprnt = ^r; flush = ^o; werase = ^w; lnext = ^v;
-parenb -parodd cs8 -cstopb hupcl cread -clocal -loblk parext
-ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl -iuclc
ixon ixany -ixoff -imaxbel
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop -echoctl echoprt -echoke -defecho -flusho -pendin -iexten
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel ff1
```

Example 2

If you have inadvertently performed an action that causes inputs not to be shown on the screen, you can restore the screen display by entering the following command:

```
$ stty echo
```

The input of the above command will, however, not be displayed.

Example 3

Reset all terminal settings to reasonable values:

```
$ stty sane
```

Example 4

The terminal settings can also be reset with the aid of the `–g` argument:

```
$ STATUS=`stty -g`
```

The above command defines a variable with the current settings of the terminal (which may differ from those

of `stty sane`). The existing terminal settings can now be changed by resetting them with:

```
$ stty $STATUS
```

SEE ALSO

`tabs`(1), `ioctl`(2), `ascii`(5), `termio`(7), `termiox`(7).

User's Reference Manual

stty(1-ucb)
BSD Compatibility Package

NAME

`stty -` set the options for a terminal

SYNOPSIS

`/usr/ucb/stty` [`–a`] [`–g`] [`–h`] [*options*]

DESCRIPTION

`stty` sets certain terminal I/O options for the device that is the current standard input; without arguments, it reports the settings of certain options.

In this report, if a character is preceded by a caret (`^`), then the value of that option is the corresponding CTRL character (for example, "`^h`" is `CTRL-h`; in this case, recall that `CTRL-h` is the same as the back-space key.) The sequence "`^-`" means that an option has a null value.

OPTIONS

`–a` Reports all of the option settings.

`–g` Reports current settings in a form that can be used as an argument to another `stty` command.

`–h` Reports all the option settings with the control characters in an easy to read column format.

Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sanity checking is performed. Hardware flow control and clock modes options may not be supported by all hardware interfaces. The options are selected from the following:

Special Requests

`all` Reports the same option settings as `stty` without arguments, but with the control characters in column format.

`everything`
Everything `stty` knows about is printed. Same as `–h` option.

`speed`
The terminal speed alone is reported on the standard output.

`size` The terminal (window) sizes are printed on the standard output, first rows and then columns. This option is only appropriate if currently running a window system.

`size` and `speed` always report on the settings of `/dev/tty`, and always report the settings to the standard output.

Control Modes

`parenb` (`–parenb`)
Enable (disable) parity generation and detection.

`parext` (`–parext`)
Enable (disable) extended parity generation and detection for mark and space parity.

`parodd` (`–parodd`)
Select odd (even) parity, or mark (space) parity if `parext` is enabled.

`cs5 cs6 cs7 cs8`
Select character size [see `termio(7)`].

`0` Hang up line immediately.

`110 300 600 1200 1800 2400 4800 9600 19200 exta 38400 extb`
Set terminal baud rate to the number given, if possible. (All speeds are not supported by all hardware interfaces.)

`ispeed 0 110 300 600 1200 1800 2400 4800 9600 19200 exta 38400 extb`
: Set terminal input baud rate to the number given, if possible. (Not all hardware supports split baud rates.) If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate.

`ospeed 0 110 300 600 1200 1800 2400 4800 9600 19200 exta 38400 extb`
: Set terminal output baud rate to the number given, if possible. (Not all hardware supports split baud rates.) If the baud rate is set to zero, the line will be hung up immediately.

`hupcl` (`–hupcl`)
: Hang up (do not hang up) connection on last close.

`hup` (`–hup`)
: Same as `hupcl` (`–hupcl`).

`cstopb` (`–cstopb`)
: Use two (one) stop bits per character.

`cread` (`–cread`)
: Enable (disable) the receiver.

`clocal` (`–clocal`)
: Assume a line without (with) modem control.

`loblk` (`–loblk`)
: Block (do not block) output from a non-current layer.

Input Modes

`ignbrk` (`–ignbrk`)
: Ignore (do not ignore) break on input.

`brkint` (`–brkint`)
: Signal (do not signal) `INTR` on break.

`ignpar` (`–ignpar`)
: Ignore (do not ignore) parity errors.

`parmrk` (`–parmrk`)
: Mark (do not mark) parity errors

`inpck` (`–inpck`)
: Enable (disable) input parity checking.

`istrip` (`–istrip`)
: Strip (do not strip) input characters to seven bits.

`inlcr` (`–inlcr`)
: Map (do not map) NL to CR on input.

`igncr` (`–igncr`)
: Ignore (do not ignore) CR on input.

`icrnl` (`–icrnl`)
: Map (do not map) CR to NL on input.

`iuclc` (`–iuclc`)
: Map (do not map) upper-case alphabetics to lower case on input.

`ixon` (`–ixon`)
: Enable (disable) START/STOP output control. Output is stopped by sending an STOP and started by sending an START.

`ixany` (`–ixany`)
: Allow any character (only START) to restart output.

`decctlq` (`–decctlq`)
: Same as `–ixany`.

`ixoff` (`–ixoff`)

Request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

`tandem` (`–tandem`)
Same as `ixoff`.

`imaxbel` (`–imaxbel`)
Echo (do not echo) BEL when the input line is too long.

`iexten` (`–iexten`)
Enable (disable) extended (implementation-defined) functions for input data.

Output Modes

`opost` (`–opost`)
Post-process output (do not post-process output; ignore all other output modes).

`olcuc` (`–olcuc`)
Map (do not map) lower-case alphabetics to upper case on output.

`onlcr` (`–onlcr`)
Map (do not map) NL to CR-NL on output.

`ocrnl` (`–ocrnl`)
Map (do not map) CR to NL on output.

`onocr` (`–onocr`)
Do not (do) output CRs at column zero.

`onlret` (`–onlret`)
On the terminal NL performs (does not perform) the CR function.

`ofill` (`–ofill`)
Use fill characters (use timing) for delays.

`ofdel` (`–ofdel`)
Fill characters are DELs (NULs).

`cr0 cr1 cr2 cr3`
Select style of delay for carriage returns.

`nl0 nl1`
Select style of delay for line-feeds.

`tab0 tab1 tab2 tab3`
Select style of delay for horizontal tabs.

`bs0 bs1`
Select style of delay for backspaces.

`ff0 ff1`
Select style of delay for form-feeds.

`vt0 vt1`
Select style of delay for vertical tabs.

Local Modes

`isig` (`–isig`)
Enable (disable) the checking of characters against the special control characters INTR, QUIT, and SWTCH.

`icanon` (`–icanon`)
Enable (disable) canonical input (ERASE and KILL processing).

`cbreak` (`–cbreak`)
Same as `–icanon`.

`xcase` (`–xcase`)
Canonical (unprocessed) upper/lower-case presentation.

`echo` (`–echo`)
: Echo back (do not echo back) every character typed.

`echoe` (`–echoe`)
: Echo (do not echo) ERASE character as a backspace-space-backspace string. This mode will erase the ERASEed character on many CRT terminals; however, it does not keep track of column position and, as a result, may be confusing on escaped characters, tabs, and backspaces.

`crterase` (`–crterase`)
: Same as `echoe`.

`echok` (`–echok`)
: Echo (do not echo) NL after KILL character.

`lfkc` (`–lfkc`)
: The same as `echok` (`–echok`); obsolete.

`echonl` (`–echonl`)
: Echo (do not echo) NL.

`noflsh` (`–noflsh`)
: Disable (enable) flush after INTR, QUIT, or SWTCH.

`stwrap` (`–stwrap`)
: Disable (enable) truncation of lines longer than 79 characters on a synchronous line.

`tostop` (`–tostop`)
: Send (do not send) `SIGTTOU` for background processes.

`echoctl` (`–echoctl`)
: Echo (do not echo) control characters as `^`*char*, e.g. DEL as "`^?`".

`ctlecho` (`–ctlecho`)
: Same as `echoctl`.

`echoprt` (`–echoprt`)
: Echo (do not echo) erase character as character is erased.

`prterase` (`–prterase`)
: Same as `echoprt`.

`echoke` (`–echoke`)
: BS-SP-BS erase (do not BS-SP-BS erase) entire line on line kill.

`crtkill` (`–crtkill`)
: Same as `echoke`.

`flusho` (`–flusho`)
: Output is (is not) being flushed.

`pendin` (`–pendin`)
: Retype (do not retype) pending input at next read or input character.

`stflush` (`–stflush`)
: Enable (disable) flush on a synchronous line after every `write()`.

`stappl` (`–stappl`)
: Use application mode (use line mode) on a synchronous line.

Hardware Flow Control Modes

`rtsxoff` (`–rtsxoff`)
: Enable (disable) RTS hardware flow control on input.

`ctsxon` (`–ctsxon`)
: Enable (disable) CTS hardware flow control on output.

`dterxoff` (`–dterxoff`)
: Enable (disable) DTER hardware flow control on input.

`rlsdxon` (`-rlsdxon`)
Enable (disable) RLSD hardware flow control on output.

`isxoff` (`-isxoff`)
Enable (disable) isochronous hardware flow control on input.

Clock Modes

`xcibrg`
Get transmit clock from internal baud rate generator.

`xctset`
Get the transmit clock from transmitter signal element timing (DCE source) lead, CCITT V.24 circuit 114, EIA-232-D pin 15.

`xcrset`
Get transmit clock from receiver signal element timing (DCE source) lead, CCITT V.24 circuit 115, EIA-232-D pin 17.

`rcibrg`
Get receive clock from internal baud rate generator.

`rctset`
Get receive clock from transmitter signal element timing (DCE source) lead, CCITT V.24 circuit 114, EIA-232-D pin 15.

`rcrset`
Get receive clock from receiver signal element timing (DCE source) lead, CCITT V.24 circuit 115, EIA-232-D pin 17.

`tsetcoff`
Transmitter signal element timing clock not provided.

`tsetcrc`
Output receive clock on transmitter signal element timing (DTE source) lead, CCITT V.24 circuit 113, EIA-232-D pin 24, clock source.

`tsetcxc`
Output transmit clock on transmitter signal element timing (DTE source) lead, CCITT V.24 circuit 113, EIA-232-D pin 24, clock source.

`rsetcoff`
Receiver signal element timing clock not provided.

`rsetcrc`
Output receive clock on receiver signal element timing (DTE source) lead, CCITT V.24 circuit 128, no EIA-232-D pin, clock source.

`rsetcxc`
Output transmit clock on receiver signal element timing (DTE source) lead, CCITT V.24 circuit 128, no EIA-232-D pin, clock source.

Control Assignments

control-character c
Set *control-character* to *c*, where *control-character* is `intr`, `quit`, `erase`, `kill`, `eof`, `eol`, `eol2`, `swtch`, `start`, `stop`, `susp`, `dsusp`, `rprnt`, `flush`, `werase`, `lnext` `min`, `ctab`, `time`, or `brk`) [`ctab` is used with `-stappl`; `min` and `time` are used with `-icanon`; see `termio(7)`]. If *c* is preceded by an (escaped from the shell) caret (^), then the value used is the corresponding CTRL character (for example, "^d" is a `CTRL-d`); "^?" is interpreted as DEL and "^-" is interpreted as undefined.

`line` *i*
Set line discipline to *i* (0 < *i* < 127).

Combination Modes

`evenp` or `parity`

Enable `parenb` and `cs7`.

`–evenp`, or `–parity`
: Disable `parenb`, and set `cs8`.

`even` (`–even`)
: Same as `evenp` (`–evenp`).

`oddp` Enable `parenb`, `cs7`, and `parodd`.

`–oddp`
: Disable `parenb` and `parodd`, and set `cs8`.

`odd` (`–odd`)
: Same as `oddp` (`–oddp`).

`spacep`
: Enable `parenb`, `cs7`, and `parext`.

`–spacep`
: Disable `parenb` and `parext`, and set `cs8`.

`markp`
: Enable `parenb`, `cs7`, `parodd`, and `parext`.

`–markp`
: Disable `parenb`, `parodd`, and `parext`, and set `cs8`.

`raw` (`–raw` or `cooked`)
: Enable (disable) raw input and output (no ERASE, KILL, INTR, QUIT, SWTCH, EOT, or output post processing).

`nl` (`–nl`)
: Unset (set) `icrnl`, `onlcr`. In addition `–nl` unsets `inlcr`, `igncr`, `ocrnl`, and `onlret`.

`lcase` (`–lcase`)
: Set (unset) `xcase`, `iuclc`, and `olcuc`.

`LCASE` (`–LCASE`)
: Same as `lcase` (`–lcase`).

`tabs` (`–tabs` or `tab3`)
: Preserve (expand to spaces) tabs when printing.

`ek` Reset ERASE and KILL characters back to normal `#` and `@`.

`sane` Resets all modes to some reasonable values.

`term` Set all modes suitable for the terminal type *term*, where *term* is one of `tty33`, `tty37`, `vt05`, `tn300`, `ti700`, or `tek`.

`async`
: Set normal asynchronous communications where clock settings are `xcibrg`, `rcibrg`, `tsetcoff` and `rsetcoff`.

`litout` (`–litout`)
: Disable (enable) `parenb`, `istrip`, and `opost`, and set `cs8` (`cs7`).

`pass8` (`–pass8`)
: Disable (enable) `parenb` and `istrip`, and set `cs8` (`cs7`).

`crt` Set options for a CRT (`echoe`, `echoctl`, and, if ≥ 1200 baud, `echoke`.)

`dec` Set all modes suitable for Digital Equipment Corp. operating systems users (ERASE, KILL, and INTR characters to "`^?`", "`^U`", and "`^C`", `decctlq`, and `crt`.)

Window Size

`rows` *n*
: Set window size to *n* rows.

`columns` *n*

Set window size to *n* columns.

`cols` *n*
: An alias for `columns` *n*.

`ypixels` *n*
: Set vertical window size to *n* pixels.

`xpixels` *n*
: Set horizontal window size to *n* pixels.

SEE ALSO

`tabs`(1), `ioctl`(2), `termio`(7), `termiox`(7).

User's Reference Manual

su(1)

NAME

su - temporarily switch to another account (become superuser oranother user)

SYNOPSIS

su [–] [*login_name* [*arg ...*]]

DESCRIPTION

su allows you to work temporarily under another login name (in another account).

Except for the system administrator (i.e. superusers), all users who invoke su are asked to supply the appropriate password for whichever account they wish to switch to. The system administrator (logged in as root) may work in any account without providing a password. If a shell is entered in the /etc/passwd file as the start-up program for the specified login name, su will execute a new subshell, using the user ID associated with the given login name as the real and effective user ID of this new shell process. If no such entry exists in /etc/passwd, the Bourne shell /usr/bin/sh is started by default. The PATH variable [see sh(1)] retains the value defined in the environment. If you switch to the system administrator's account, PATH is set to the value of the SUPATH variable from /etc/default/su. For the rest, the new subshell inherits the original environment of the user who calls su, for example $USER and $HOME. If you wish to change the environment to what would be expected if you had actually logged in as the specified *login_name*, you can do this by calling su with the – option.

All system-wide attempts to become another user with su (including those made from remote systems) are logged in the file /var/adm/sulog (if the environment variable SULOG is defined in /etc/default/su); all such attempts to switch to the system administrator's account are output on the console (if the environment variable CONSOLE is defined in /etc/default/su).

The invoked subshell can be exited by pressing the <CTRL-D> key. You will then be returned to your original account with your previous environment exactly as it was when you left it.

Checking your current login name

The id(1) command displays the login name under which you are currently identified.

Checking your original login name

The logname(1) and who(1) commands display the login name under which you last logged in with the login command.

OPTIONS

– If you use the dash option (–), and the start-up program named in the file /etc/passwd for the user *login_name* is sh, your environment will be changed to what would be expected if you had originally logged in (with login) as *login_name*. A login shell is invoked [see login(1)] and the system's profile (/etc/profile) and then the user-specific profile ($HOME/.profile) are executed. Your .profile can check *arg0* for –sh or –su to determine if it was invoked by login or su, respectively.

If the start-up program is a *program* other than /usr/bin/sh, –*program* is passed to .profile as *arg0*, regardless of whether it was invoked by login or su.

login_name

Login name under which you wish to work temporarily. You will be asked for the password of *login_name* before you can proceed further.

Only the system administrator can work in other accounts without giving the password.

login_name not specified:

As system administrator (superuser) you assume the privileges of root (after giving the appropriate password!).

arg One or more arguments to be passed to the startup program entered for the specified login name in the `/etc/passwd` file (the command interpreter `/bin/sh` by default). These arguments can be specified on the `su` command line exactly as for the command interpreter.

If the startup program is a shell, it is also possible to specify a single argument of the form `-c` *command* or `-r`:

`-c` *command*
Executes *command* in the start-up shell with the environment of the specified login name.

`-r` Calls a restricted shell.

ERROR MESSAGES

`su: Sorry`
You have entered an invalid login name or password.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

These examples illustrate the difference between the `login` command and the `su` command.

Example 1

Use the command (`login`) to log in to the system under the login name (`guest`); then check the current login name (with `id`), the original login name (with `who am i`), and the value of the environment variable `LOGNAME` (with `echo $LOGNAME`):

```
$ login guest
Password: blanked input of the password for guest
$ id
uid=100(guest) gid=10(other)
$ who am i
guest     grtty   MAR 17 15:13
$ echo $LOGNAME
guest
```

Example 2

Use `su` without the – option to switch to the login name `franny`. Although `franny`'s environment redefines the `PS1` variable (i.e. the shell prompt) to the value FR, this variable will not be evaluated. Check the current login name (with `id`), the original login name (with `who am i`), the value of the environment variable `LOGNAME` (with `echo $LOGNAME`), and return to the login name `guest` (with <CTRL-D>):

```
$ su franny
Password: blanked input of the password for franny
$ id
uid=103(franny) gid=10(other)
$ who am i
guest     grtty   MAR 17 15:13
$ echo $LOGNAME
guest
$ <CTRL-D>$ id
uid=100(guest)  gid=10(other)
```

Example 3

Use `su`, this time with the – option, to switch to the login name `franny`. The environment variable `PS1` (i.e. the shell prompt), which is redefined in `franny`'s environment to FR, will now be evaluated. Check the current

login name (with id), the original login name (with who am i) and the value of the environment variable LOGNAME (with echo $LOGNAME), then execute logname and finally return to the login name guest (with <CTRL-D>).

```
$ su – franny
Password: blanked input of the password for franny
FR id
uid=103(franny) gid=10(other)
FR who am i
guest      grtty   MAR 17 15:13
FR echo $LOGNAME
franny
FR logname
guest
FR <CTRL-D>$
```

Example 4

Use su –c to switch to the login name franny and, in the temporary environment and with the permissions of user franny, execute the command id (which displays the current login name):

```
$ su – franny –c "id"
Password: blanked input of the password for franny
uid=103(franny) gid=10(other)
$
```

ENVIRONMENT VARIABLES

SULOG
: If a *file_name* is assigned to this variable in /etc/default/su, all attempts to switch to another account with su are logged in *file_name*. The default value for SULOG is /var/adm/sulog.

CONSOLE
: If a terminal name is assigned to this variable in /etc/default/su, all attempts to use su root are logged on that terminal. The default value for CONSOLE is /dev/console.

PATH
: The value of this variable in /etc/default/su is assigned to the PATH variable if you switch accounts using su –.

SUPATH
: If you switch to the system administrator's account, the PATH environment variable is assigned the value of the SUPATH variable from /etc/default/su.

FILES

/etc/passwd
: System's login and password file.

/etc/profile
: System profile executed for each login name by the login shell.

$HOME/.profile
: Private shell script that every user can set up in his or her home directory. This profile is automatically executed by the login shell after /etc/profile.

/var/adm/sulog
: File that logs every attempt made in the system to become another user, provided the SULOG variable is set to /var/adm/sulog.

/etc/default/su
: File containing the default values for the variables SULOG, CONSOLE, PATH and SUPATH, e.g. SULOG is set to /var/adm/sulog and CONSOLE is set to /dev/console.

SEE ALSO

env(1), login(1), sh(1), passwd(4), profile(4), su(4), environ(5).

User's Reference Manual

sum(1)

NAME

`sum` - print checksum and block count of a file

SYNOPSIS

`sum [ –r ] [ –– ] [` *file ...* `]`

DESCRIPTION

`sum` calculates and prints a checksum for the named file or directory, and also prints the space used by the file, in 512-byte blocks. The output is written to standard output. `sum` counts the contents of all bytes together for calculating the checksum.

`sum` may be useful for checking that a file has arrived complete and unchanged after a file transfer.

OPTIONS

`–r` Uses an alternate algorithm to compute the checksum and prints a different checksum. When the checksum is being calculated, the sum is shifted 1 bit to the right before the next byte is added.

`––` End of the list of options. Must be specified if *file* begins with –.

file Name of the file whose checksum is to be calculated. Both ordinary files and directories may be specified. You may name more than one file.

file not specified:

`sum` reads from standard input.

Caution:

The algorithms used in computing the checksum may not be portable. Different checksums may thus be produced for the same file when `sum` is run on different systems.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Checksum of the file `months` using the first algorithm:

```
$ sum months
6658 1 months
```

Example 2

Checksum of the file `months` using the second algorithm:

```
$ sum -r months
62412     1 months
```

NOTES

The following error message is displayed if *file* is the mount point for an NFS file system:

`sum: Read error on` *file*`: is a directory`

SEE ALSO

`cksum`(1), `md5`(1), `wc`(1).

User's Reference Manual

sum(1-ucb)
BSD Compatibility Package

NAME

`sum -` calculate a checksum for a file

SYNOPSIS

`/usr/ucb/sum` *file*

DESCRIPTION

`sum` calculates and displays a 16-bit checksum for the named file or directory, also displays the size of the file in kilobytes. It is typically used to look for bad spots, or to validate a file communicated over some transmission line. The checksum is calculated by an algorithm which may yield different results on machines with 16-bit `int`s and machines with 32-bit `int`s, so it cannot always be used to validate that a file has been transferred between machines with different-sized `int`s.

DIAGNOSTICS

`Read error` is indistinguishable from EOF on most devices; check the block count.

NOTES

The following error message is displayed if *file* is the mount point for an NFS file system:

`sum: Read error on` *file*

SEE ALSO

`cksum(1)`, `sum(1)`, `wc(1)`.

tabs(1) - vi(1)

User's Reference Manual

tabs(1)

NAME

`tabs -` set tabs on a terminal

SYNOPSIS

`tabs` [`–T` *type*] [`+m`*n*] [*tabspec*]

DESCRIPTION

`tabs` sets the tab stops on the user's terminal, clearing any previous settings. For `tabs` to be effective, the terminal must have hardware tabs that can be set remotely using escape sequences.

The lowest column number is 1. For `tabs`, column 1 refers to the leftmost column on the screen, even on terminals whose column markers begin at 0.

OPTIONS

`–T` *type*

Terminal type, which needs to be known for margin setting. *type* is a name listed in `/usr/share/lib/terminfo` [see `term(4)`].

`–T` *type* not specified:

`tabs` uses the value of the `TERM` environment variable. If `TERM` is not defined in the current environment [see `environ(5)`], `tabs` uses a type whose attributes are valid for many terminals.

`+m`*n* Resets the left margin. All tabs are moved *n* columns to the right, with *n*+1 becoming the left margin. If *n* is omitted, a value of 10 is assumed. For a TermiNet, the first value in the tab list should be 1, or the margin will move even further to the right. Normally, the leftmost margin is set with `+m0`. On most terminals, the margin is reset further to the right only when `+m` is specified explicitly.

tabspec

Four different types of tab specification are permitted:

–code

tabs in a predefined pattern (*canned*)

–n tabs at regular intervals (*repetitive*)

––file

tabs in a pattern defined in *file* (*file*)

n1,n2 ...

tabs at freely selectable positions (*arbitrary*)

–code

You determine the pattern of tabs on a line by specifying one of the following:

`–a` Tab stops in columns 1, 10, 16, 36, 72.

This corresponds to Assembler, IBM S/370, first format.

`–a2` Tab stops in columns 1, 10, 16, 40, 72.

This corresponds to Assembler, IBM S/370, second format.

`–c` Tab stops in columns 1, 8, 12, 16, 20, 55.

This corresponds to COBOL normal format.

`–c2` Tab stops in columns 1, 6, 10, 14, 49.

This corresponds to COBOL compact format. Columns 1-6 are omitted. The first character typed corresponds to column 7, one space takes you to column 8, and a tab reaches column 12. Files using this tab setup should include a format specification as follows [see `fspec(4)`]: `<:t-c2m6s66d:>`.

`–c3` Tab stops in columns 1, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, 67.

This corresponds to COBOL compact format with additional tabs. Columns 1-6 are omitted. The first character typed corresponds to column 7, one space takes you to column 8, and a tab reaches column 12. Files using this tab setup should include a format specification as follows [see `fspec(4)`]: `<:t-c3m6s66d:>`.

This is the recommended format for COBOL.

`–f` Tab stops in columns 1, 7, 11, 15, 19, 23.

This corresponds to FORTRAN format.

`–p` Tab stops in columns 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61.

This corresponds to PL/I format.

`–s` Tab stops in columns 1, 10, 55.

This corresponds to SNOBOL format.

`–u` Tab stops in columns 1, 12, 20, 44.

This corresponds to UNIVAC 1100 Assembler format.

–n The number specified causes tab stops to be set in columns 1+*n*, 1+2*n*, 1+3*n*. Specifying 8 tabs gives you the UNIX system standard tab setting. Specifying 0 clears all tab stops.

––file

`tabs` reads the first line of the file, searching for a format specification [see `fspec(4)`]. If it finds one, it sets the tab stops accordingly. Otherwise, the default value `–8` is assumed. This type of format may be used to ensure that a tabbed file is printed with the correct tab settings,

with `tabs` being used in conjunction with `pr` [see `pr(1)`].

n1,*n2*, ...

You can freely select the columns for the tab stops by means of a list containing up to 64 numbers in ascending order. A number preceded by a plus sign is added to the value of the previous number. This does not apply to the first number. The format `1,10,20,30`, for example, can be specified as `1,10,+10,+10`.

The numbers are separated by a comma, or the list is enclosed in quotes, whereby the numbers are separated by a comma and/or a blank. *n1*,*n2*, ... must be specified as the last argument in the command line.

`tabspec` not specified:

`tabs` uses the default setting `-8`, which corresponds to the UNIX system standard tab setting.

Note:

The mechanisms for clearing tab stops and setting the left margin are not the same on every terminal.

`tabs` can set a maximum of 64 tab stops, but will only clear 20.

ERROR MESSAGES

`illegal tabs`

When setting arbitrary tab stops you failed to maintain ascending order or you specified a zero.

`illegal increment`

When setting arbitrary tab stops you specified a zero or omitted an increment.

`unknown tab code`

The code you specified as *tabspec* cannot be found.

`can't open`

The file you specified as *tabspec* cannot be opened.

`file indirection`

The format specification in the file specified as *tabspec* points to another file. Indirect references of this sort are not permitted.

ENVIRONMENT VARIABLES

`TERM` Terminal type.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Setting tab stops in COBOL normal format:

```
$ tabs -c
```

Example 2

Set a tab in every sixth column, i.e. in columns 1, 7, 13, 19, ...:

```
$ tabs -6
```

Example 3

Set tabs in columns 1, 8 and 36:

```
$ tabs 1,8,36
```

This has the same effect as:

```
$ tabs 1,+7,+28
```

Example 4

Setting tab stops in accordance with the first line (format specification) of your file `$HOME/tabspec.list/file1`:

```
$ tabs --$HOME/tabspec.list/file1
```

FILES

`/usr/share/lib/terminfo/?/*`
: Files used to name and define terminals

SEE ALSO

`expand`(1), `newform`(1), `pr`(1), `stty`(1), `tput`(1), `unexpand`(1), `fspec`(4), `term`(4), `terminfo`(4), `environ`(5), `term`(5), `term_names`(5).

User's Reference Manual

tail(1)

NAME

`tail -` deliver the last part of a file

SYNOPSIS

Format 1

`tail` [`–f`] {`–c` *number* | `–n` *number*} [`––`] [*file*]

Format 2

`tail` [*place*[`bcl`[`rf`]]] [*file*]

DESCRIPTION

`tail` prints the contents of the specified file on standard output beginning at a designated place. If no file name is specified, `tail` outputs the contents of standard input.

Tails relative to the end of the file are saved in an internal buffer, and thus are limited in length. Such a buffer is no smaller than `LINE_MAX`*10 bytes [see `limits(4)`].

Various kinds of anomalous behavior may occur if `tail` is used on character special files.

OPTIONS

Format 1: New synopsis for tail

`–c` *number*

Designates the place in the input file, in bytes, where output is to begin. *number* can be specified signed or unsigned:

`+` Counting starts at the beginning of the file.

`–` Counting starts at the end of the file.

Unsigned
Counting starts at the end of the file.

`–c` must not be combined with `–n`.

Neither `–c` nor `–n` specified: The last 10 lines of *file* are output (corresponds to `–n 10`).

`–f` (f – follow) If the input file is a file or a FIFO file, the program will not terminate after the last line of output has been displayed, but will enter an endless loop, wherein it sleeps for at least a second and then attempts to read and copy further records from the input file. You could use this option to monitor the growth of a file that is being written by another process. If the input file is a pipe, `–f` is ignored.

`–n` *number*

Designates the place in the input file, in lines, where output is to begin. *number* can be specified signed or unsigned (see `–c`).

`–n` cannot be combined with `–c`.

Neither `–n` nor `–c` specified:

The last 10 lines of *file* are output (corresponds to `–n 10`).

`–n` *number* corresponds to the old option *–number* if *number* is unsigned or negative (with minus sign), and corresponds to the old option `\+`*number* if *number* is positive (with plus sign).

`––` End of the list of options. Must be specified if *file* begins with `–`.

file Name of the input file to be output by `tail`.

file not specified: `tail` reads from the standard input.

Format 2: Old synopsis for tail

place

place defines the position in the input file at which display is to begin. This location may be given as:

+[*number*]
–[*number*]

number can be specified as any decimal integer value. +[*number*] is calculated relative to the beginning of the file; –[*number*] is calculated relative to the end of the file.

number not specified:

The default value of 10 is assumed.

place not specified:

The default value for *place* is –10, i.e. 10 lines from the end of the file.

`b` Starts display at the block defined by *place*. A block is a 512-byte unit.

`c` Starts display at the character defined by *place*. (Corresponds to the –`c` option in the new synopsis.)

`l` Starts display at the line defined by *place*. (Corresponds to the –`n` option in the new synopsis.)

Only one of the options `b`, `c`, or `l` may be used at a time. If you omit all three options, the `l` option is set by default.

`f` (follow) If the input file is a file or a FIFO file, the program will not terminate after the last line of the input file has been displayed, but will enter an endless loop, wherein it sleeps for at least a second and then attempts to read and copy further records from the input file. You could thus use this option to monitor the growth of a file that is being written by some other process. If the input file is a pipe, `f` is ignored. (Corresponds to the –`f` option in the new synopsis.)

`r` (reverse) Lines are copied from the specified starting point in the file in reverse order. The default is to output the entire file in reverse order.

If the`r` option is used together with a numeric value [*number*], the numeric value must be negative. Thus in this case [*number*] must not be combined with +.

You must not combine the `f` and `r` options.

file Name of the input file to be displayed by `tail`.

file not specified: `tail` reads from standard input.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The file `months` contains the name of a month in each line. The following two calls to `tail`:

```
$ tail -n 5 months
$ tail -n +8 months
```

will result in the same output:

```
August
September
October
November
December
```

Example 2

To observe the effect of the `-f` option, create a shell script that writes data to a file in an endless loop and run this script in the background.

The shell script `neverend` with the following contents would serve the purpose:

```
while true
do {
   date >>anne
   sleep 5
   }
done
```

You can now run this script in the background and then call `tail` with option `-f` and the file `anne` as its argument:

```
$ neverend &
656
$ tail -f -n 5 anne
These five
lines
were already
in the file
before any additions.
FRI MAR 22 11:40:35 MET 1991
FRI MAR 22 11:40:40 MET 1991
.
.
.
```

`tail` first displays the last five lines of the file `anne`, followed by the data written to this file by the script `neverend`. The process generated with `tail -f` can be terminated by pressing the CTRL-C key.

SEE ALSO

`cat(1)`, `head(1)`, `more(1)`, `pg(1)`, `limits(4)`.

User's Reference Manual

tar(1)

NAME

`tar -` archive files and manipulate archives (tape archiver)

SYNOPSIS

Format 1

`tar` *option*[*modifier*] *file ...*

Format 2

`tar t`[*modifier*] [*file ...*]

Format 3

`tar x`[*modifier*] [*file ...*]

DESCRIPTION

`tar` can be used to:

- archive files or directories on data storage media (**Format 1**).
- list the contents of an archive (**Format 2**).
- extract all or part of an archive created with `tar` (**Format 3**).

The system administrator may use `tar` to back up and restore all files and directories. Non-privileged users can only back up files and directories for which they have access permission (read permission for files and execute permission for directories). Non-privileged users wanting to restore archived files and directories require write permission for the directory to be written to.

Note:

`tar` archives can also be read and created using the `pax(1)` and `cpio(1)` commands.

Functionality

`tar` processes files in archival form on a data storage medium. Only one archive can be processed with a single invocation of `tar`.

It is not possible to create more than one archive on a floppy disk.

By contrast, you can create several archives on a magnetic tape or tape cartridge and use the command `mt(1)` to wind to the start or end of a particular archive. `tar` always reads from the archive at the current tape position. If all of several archives are to be read from a storage medium, a separate `tar` command must be given for each archive.

If you use the appropriate special file when extracting an archive, i.e. a special file without automatic rewinding, the tape position will automatically be set to the start of the next archive after an archive is read. The tape need not be repositioned with `mt(1)` in such a case.

Before the call

Before you execute `tar`, you must first load the storage medium in the appropriate drive. If you wish to create a new archive, the write protect mechanism used on the device must be removed. See the operating manual provided with your computer for more information. If the archive is to be created on a floppy disk, the floppy disk will need to be formatted. The `dosformat(1)` or `flformat(1M)` command is used for this purpose.

After the call

It is advisable to remove the data medium from the drive after executing `tar`. Otherwise, another user might call `tar` again and inadvertently destroy or corrupt your archive.

In each of the three formats the main option can be combined with various modifiers.

Format 1: Archiving files and directories

`tar` *option*[*modifier*] *file ...*

option

[–]`c`, [–]`r`, [–]`u`

modifier without argument

`A`, `e`, `L`, `n`, `v`, `w`, *number*

modifier with argument

`b` *blocking-factor*, `f` *archive*, `F` *ffile*, `k` *kbytes*, `X` *xfile*

If you are a normal user without system administrator privileges, you may only archive files and directories for which you have access permission. Read permission is required for files; execute permission for directories.

Options

option is any one of the following letters, with or without a minus sign:

[–]`c` *Floppy disks*:

(create) `tar` creates a new archive on the floppy disk and writes the specified files at the beginning of this archive, overwriting any archive previously existing on the floppy disk in the process. If you specify more than one file, they are entered sequentially. If you specify a directory, all subtrees of the directory are archived recursively.

Magnetic tapes and tape cartridges:

`tar` writes the specified file or directory at the currently set position of the storage medium. You can use the `mt(1)` command to wind on to the end of an archive. Thus `tar c` allows you to append several archives to one another. Note that you can prevent the tape from being automatically rewound after being accessed by the `tar` command by specifying a "no rewind" special file.

If there are several archives on a streamer tape, overwriting any one of them will render all subsequently stored archives unreadable.

Caution:

If you need to archive more data than will fit on a single volume, you must initialize continuation volume processing using the `k` modifier.

[–]`r` This function applies only to floppy disk or hard disk archives.

The named files are appended to the end of an existing archive. The appended files belong to the same archive.

If there are a number of files with the same name in an archive, restoring files from that archive will produce only one instance of a file with that name, earlier instances being successively overwritten.

You can only use this option in conjunction with the `n` option.

[–]`u` (update) This function applies only to floppy disk or hard disk archives.

The named files are appended to the end of an existing archive only if

- they do not already exist in the archive
- or they have a more recent modification date than the archived files.

The appended files belong to the same archive.

If there are a number of files with the same name in an archive, restoring files from that archive will produce only one instance of a file with that name, earlier instances being successively overwritten.

You can only use this options in conjunction with the `n` option.

Modifiers

The effect of the selected function can be influenced by specifying function modifiers. These modifiers must be appended to the option letter without intervening blanks and can in some cases also take arguments. The following points must be observed:

1. All modifiers must be grouped together without blanks, followed by
2. all arguments, separated by blanks. The arguments must be in the same order as the modifiers with

which they are associated.

`A` If you specify the files to archive in the form of absolute path names, `tar` suppresses the first slash.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`b` *blocking_factor*

The blocking factor specifies the number of tape blocks to be written by `tar` per tape or floppy disk access.

blocking_factor

Any number from 1 to 20 can be given as the blocking factor.

`b` *blocking_factor* not specified: Blocking factor = 1.

`e` Errors cause `tar` to abort immediately.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`f` *archive*

`tar` expects you to specify an archive to write to.

archive may be one of the following:

special_file

Name of the special file to which the archive is to be written.

`–` (dash) If you use a dash as *argument*, `tar` writes to standard output. This feature enables `tar` to be used in a pipe to copy or move directories or file systems (see *Example 5*).

file If you specify a regular file as *archive*, a `tar` archive named *archive* is created on the hard disk.

`f` *archive* not specified:

If the `TAPE` shell variable is set, `tar` takes the value of `TAPE` as the current special file. If `TAPE` is not set, tar selects the file associated with 0 in the `/etc/default/tar` file.

`F` *ffile*

`tar` reads the names of the files to archive from the file *ffile*. *ffile* is a list of file names, one name per line. Any other files you want to archive can be named on the command line in the usual way.

This modifier cannot be combined with the `X` modifier.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`k` *kbytes*

If you need to archive more data than will fit on **a single** volume, you must initialize continuation tape processing using the `k` modifier.

kbytes is the length of the volume to be used in kilobytes, minus a safety margin in the case of tapes. During writing, any bad spots on the tape will automatically be skipped, so less data will fit on the tape. Thus if you want to use a 155K tape, you could use a *kbytes* value of 140. You should choose the size of the safety margin empirically on the basis of previous experience with the tapes you use.

Once *kbytes* kilobytes have been written to the volume, you will be requested to insert another volume and press RETURN.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`L` (links) `tar` follows symbolic links. If this modifier is not specified, `tar` writes the symbolic link only, and not the corresponding files.

`n` (not tape) This modifier specifies that the argument in `f` *archive* refers only to to a medium on which `lseek(2)` can be used for positioning (hard disk, floppy disk). You must use this modifier if you want to use the `r` or `u` options.

`v` (v – verbose) Activates verbose mode, in which `tar` outputs the following information for each file it processes:

`a` (append) the specified file has been added to the archive.

path_name
the path name under which the file was entered.

n `blocks`
the number of blocks occupied by the file.

The size of a block depends on the medium used.

`v` not specified: `tar` operates silently, displaying no messages.

`w` (what) Confirmation mode; `tar` requests confirmation for each file before archiving it. The following is displayed:

option filename **:**

You are expected to enter your answer after the colon. `tar` only archives the file if you answer the locale's equivalent of `yes`.

`w` not specified: `tar` operates without requesting confirmation.

`X` *xfile*
`tar` excludes the files listed in *xfile* from the backup. This applies even if such files are also specified on the command line (either explicitly or as a result of file name expansion). *xfile* is a list of file names, one name per line.

This modifier cannot be combined with the `F` modifier.

Note:
This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

number
Here you can enter a number between 0 and 9. `tar` then selects the corresponding entry from the `/etc/default/tar` file as the special file. In `/etc/default/tar` the system administrator can list ten special files that you can then address in abbreviated form via `number`, e.g. `tar tv5`.

If the `TAPE` shell variable is set, tar uses its value as the special file.

file Names of the files or directories to be archived. You must give at least one name.

If the specified file is a directory, `tar` recursively archives this directory and all the files and subdirectories contained in it.

Shell file name generation metacharacters may also be used [see `specialchar(5)`].

Format 2: Listing the contents of an archive

`tar t`[*modifier*] [*file ...*]

modifier
`f` *special_file*, `v`, *number*

Option

`t` (table) `tar` lists the name of the file specified if it exists in the archive. If no file is specified, `tar` lists the contents of the entire archive.

Modifiers

The effect of the `tar t` command can be influenced by specifying one or more function modifiers. These modifiers must be appended to the option letter without intervening blanks. The following points must be observed when using more than one modifier:

1. All modifiers (including `f`) must be grouped together without blanks, followed by
2. *special_file* (with `f`), separated by blanks.

`f` *special_file*

`tar` expects you to specify the special file it is to read from. You also have the option of using the default special file.

special_file

Name of the special file to read from.

`f` *special_file* not specified:

`tar` selects entry 0 from the `/etc/default/tar` file (see *number* modifier). If the `TAPE` shell variable is set, `tar` takes the value of `TAPE` as the current special file.

`v` (verbose) Activates verbose mode, in which `tar` outputs the following information for each file:

- the access rights
- user and group IDs (UID/GID)
- size in bytes
- date and time the file was created
- name of the file.

`v` not specified: Only the names of the files are output.

number

Here you can enter a number between 0 and 9. `tar` then selects the corresponding entry from the `/etc/default/tar` file as the special file. In `/etc/default/tar` the system administrator can list ten special files that you can then address in abbreviated form via *number*, e.g. `tar tv5`.

If the `TAPE` shell variable is set, tar uses its value as the special file.

file Name of the file to be listed if in the archive. Several files may be specified at the same time.

If the specified *file* is a directory, `tar` recursively lists the contents of this directory and all the files and subdirectories contained in it.

file not specified: `tar` lists the contents of the entire archive.

Format 3: Extracting files

`tar x`[*modifier*] [*file ...*]

modifier without argument

`A`, `e`, `i`, `l`, `m`, `o`, `p`, `v`, `w`, `C`, *number*

modifier with argument

`f` *archive*, `F` *ffile*, `X` *xfile*, `I` *include_file*

Option

`x` Extracts the specified file from the archive. If the file was archived under its relative path name, it is restored in the current directory; if the absolute path name is in the archive, the file is restored in the appropriate directory, if present (but also refer to modifier `A`).

If you are a normal user without system administrator privileges, you will need write permission for the directory in which the file is to be restored.

If the specified file is a directory, all the files and subdirectories contained in it are restored.

If a given archived file does not exist in the directory in which it is to be restored, the file is created.

The extracted files and directories are given the user ID and group ID of the user who called `tar x` (but also refer to modifier `o`).

Access permissions are set as defined by `umask` (but also refer to modifier `p`).

If the specified file already exists in the directory in which it is to be restored, its access modes are not changed. The s bit is not taken into account unless `tar` is invoked by the system administrator. The

owner, group, and modification time remain unaltered.

If there are a number of files with the same name in an archive (see **Format 1**, options `r` and `u`), restoring files from that archive will produce only one instance of a file with that name, earlier instances being successively overwritten. The modification time is ignored in this case.

Modifiers

The effect of the `tar x` command can be influenced by specifying function modifiers. These modifiers must be appended to the option letter without intervening blanks. The following points must be observed:

1. First group together all modifiers without separating blanks.
2. Then list all the arguments, separated by blanks. The order of the arguments is determined by the order of the associated modifiers.

`A` If you specify the files to restore in the form of absolute path names,or if the files are archived under absolute path names, `tar` suppresses the first slash and restores the files to the current directory.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`e` Errors cause `tar` to abort immediately.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`f` *archive*

`tar` expects you to specify an archive to read.

archive may be one of the following:

special_file

Name of the special file to read.

`–` (dash) If you use a dash as *argument*, `tar` reads from standard input. This feature enables `tar` to be used in a pipe, for example to copy or move directories or file systems (see *Example 5*).

file If you specify a regular file as *archive*, a `tar` archive named *archive* is read from the hard disk.

`f` *archive* not specified:

If the `TAPE` shell variable is set, `tar` takes the value of `TAPE` as the current special file. If `TAPE` is not set, tar selects the file associated with 0 in the `/etc/default/tar` file.

`F` *ffile*

`tar` reads the names of the files to extract from the file *ffile*. *ffile* is a list of file names, one name per line. Any other files you want to extract can be named on the command line in the usual way.

This modifier cannot be combined with the `X` modifier.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`i` `tar` attempts to continue extracting files from the archive in spite of a bad file header checksum.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`l` `tar` reports when it cannot resolve all links to files being archived. This does not apply to symbolic links.

`l` not specified: No error messages are issued if a link cannot be resolved.

`m` (modify) Instructs `tar` to set the modification time of each file extracted from the archive to the date and time of extraction.

`m` not specified: The modification time remains the same as in the archive.

`o` (ownership) This option only works when selected by a user with system administrator privileges.

Files extracted from the archive take on the UID and GID of `root`.

`o` not specified:

An extracted file takes on the user and group IDs that appear in the archive.

`p` The extracted files are given the same permissions as in the archive, irrespective of how `umask` is set.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`v` (verbose) Activates verbose mode, in which `tar` outputs the following information for each file or directory it extracts:

`x` The specified file has extracted and placed in the appropriate directory.

k `byte`

The size of the file is *k* bytes.

n `tape blocks`

The file occupies *n* blocks. The block size depends on the medium used.

`v` not specified: `tar` operates without reporting what it is doing.

`w` (what) Confirmation mode; `tar` requests confirmation for each file before extracting it. The following is displayed:

`x` *filename*:

You are expected to enter your answer after the colon. `tar` only performs the function if you answer the locale's equivalent of `yes`.

`w` not specified: `tar` extracts the files without requesting confirmation.

`X` *xfile*

`tar` excludes the files listed in *xfile*. This applies even if such files are also specified on the command line (either explicitly or as a result of file name expansion). *xfile* is a list of file names, one name per line.

This modifier cannot be combined with the `F` modifier.

Note:

This modifier is a Reliant UNIX specific functional enhancement of the `tar` command. There is no guarantee of long-term support for this enhancement in future product releases.

`I` *include_file*

(I – Include) Specifies a file *include_file* containing a file list that is to be included in the list of files to be extracted. As with the `F` and `X` options, specify one file per line in the list.

`C` (C – Compatibility) This option is used for compatibility reasons to create archives for systems which use signed char checksum computing (e.g. the most Intel systems).

number

Here you can enter a number between 0 and 9. `tar` then selects the corresponding entry from the `/etc/default/tar` file as the special file. In `/etc/default/tar` the system administrator can list ten special files that you can then address in abbreviated form via *number*, e.g. `tar tv5`.

If the `TAPE` shell variable is set, `tar` takes the value of `TAPE` as the current special file.

file Name of the file to be extracted if in the archive. Several blank-separated files may be specified at the same time.

If the specified file is a directory, `tar` recursively extracts this directory and all the files and subdirectories contained in it.

If the file name includes shell metacharacters, they must be enclosed in single quotes '...'. Otherwise, the shell tries to substitute the metacharacters specified on the basis of the file names in the current

directory rather than those in the archive.

file not specified: `tar` extracts the entire archive.

MESSAGES

`Warning: just archiving file` *filename* `> 2GB.`
`cannot open /dev/rts0`
: You have specified a special file that is not present. This error message is also issued if you have specified the wrong special file for a data medium.

`tape write error`
: There are three types of error that can cause this error message:
- You are using an unformatted floppy disk. Use `dosformat(1)` or `flformat(1M)` to format the disk.
- If this message appears when `tar` has already started, it is likely that the specified files will not fit on the selected medium. In this case the last few files have generally not been correctly backed up.
- The data medium is write-protected.
- The data medium is defective.

`tape read error`
- Your data medium is empty.
- There is no data medium in the drive.
- The data medium is defective.

`directory checksum error`
- Your data medium is empty.
- A file has been stored incorrectly.
- The archive has been corrupted.

`blocked tapes cannot be updated`
: `r` or `u` has been specified without `n`.

`device argument required`
: No special file has been specified.

ENVIRONMENT VARIABLES

`TAPE` defines a special file to be used if the `f` modifier is not specified.

`TZ` defines the time zone within date and time specifications.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

Answers to yes/no queries must be given in the language appropriate to the current locale.

The `LC_TIME` environment variable governs the format of date and time strings in archive table-of-contents listings produced by the `v` option.

If `LC_MESSAGES` or `LC_TIME` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

The ISO 8859-1 character set [see `meta-ascii(5)`] is used for file transfers between X/Open-compliant systems. 8-bit data and file names are subject to certain restrictions in respect of porting to non-internationalized systems. Thus when transferring data between systems of different types it is advisable to use only 7-bit ASCII characters [from the ISO 646 character set, see `ascii(5)`]. When choosing file names make sure that you only use characters which are in the Portable Filename Character Set.

EXAMPLES

1. *Archiving on floppy disk*

All files contained in the current directory and its subdirectories are to be backed up on a floppy disk; the name of each file being archived is to be displayed. Any previous archive on the floppy disk is overwritten.

```
$ tar cvf /dev/ios0/sfdisk0053ht *
a  john/letters/private/aunt  9 blocks
a  john/letters/private/uncle  6 blocks
a  john/letters/job/boss1  3 blocks
a  john/book/manuscript  158 blocks
a  john/book/errors  16 blocks
a  john/tools/program  20 blocks
  ...
```

2. *Archiving on tape cartridge*

The directory `documents` is to be backed up in the second archive on a tape cartridge with a blocking factor of 20. The tape has already been positioned. Since it is not to be rewound after tape access, the special file `/dev/ios0/rstape004n` is used.

```
$ tar cbf 20 /dev/ios0/rstape004n documents
```

3. *Archiving on open-reel tape*

The file `letter` is to be archived on a tape with a blocking factor of 20 and a recording density of 1600 bpi. The tape is then to be rewound.

```
$ tar cfb /dev/ios0/rstape004 20 letter
```

4. *Listing the contents of all archives on a tape*

The contents of an entire tape are to be listed in detailed format. Since three archives are stored on the tape, the special file without rewinding is used, and `tar` is called a number of times.

```
$ tar tvf /dev/ios0/rstape004n; tar tvf /dev/ios0/rstape004n; tar tvf
/dev/ios0/rstape004n
tar: blocksize = 20
rw------- 33/1   2024   Sep 19  15:23 1988 john/letters/private/uncle
rw-rw---- 33/1   1365   Oct 20  08:12 1988 john/letters/job/boss1
  ...
tar: blocksize = 20
rw-r--r-- 45/3   2345   Jan 18  13:20 1989 patty/test
rwxr-xr-x 45/3    800   Jan 27  12:50 1989 patty/program
  ...
tar: blocksize = 20
rw------- 40/3   4567   Apr 10  07:58 1988 frank/texts/chap1
rw------- 40/3   2367   Apr 10  08:50 1988 frank/texts/chap2
  ...
```

5. *Using* `tar` in a pipeline

The directory `/home1/john` is to be copied to `/home/john`.

```
$ (cd /home1/john; tar cf - .) | (cd /home/john; tar xf -)
```

Caution:

Do not use the `v` option here, as it writes to standard output.

FILES

`/dev` Directory containing all available special files.

`/etc/default/tar`

File containing a list of up to ten special files that can be selected using the *number* modifier.

If the `TAPE` shell variable is set, `tar` takes the value of `TAPE` as the current special file.

SEE ALSO

ar(1), cpio(1), dosformat(1), gettypes(1), ls(1), mt(1), pax(1), flformat(1M).

User's Reference Manual

tcopy(1)
BSD Compatibility Package

NAME

`tcopy` - copy a magnetic tape

SYNOPSIS

`/usr/ucb/tcopy` *source* [*destination*]

DESCRIPTION

`tcopy` copies the magnetic tape mounted on the tape drive specified by the *source* argument. The only assumption made about the contents of a tape is that there are two tape marks at the end.

When only a source drive is specified, `tcopy` scans the tape, and displays information about the sizes of records and tape files. If a destination is specified, `tcopy` copies the source tape onto the *destination* tape, with blocking preserved. As it copies, `tcopy` produces the same output as it does when only scanning a tape.

NOTES

`tcopy` will only run on systems supporting an associated set of `ioctl` requests.

SEE ALSO

`mt`(1), `ioctl`(2).

User's Reference Manual

tee(1)

NAME

`tee` - join pipes and make copies of input

SYNOPSIS

`tee [ –a ] [ –i ] [ –– ] [` *file ...* `]`

DESCRIPTION

`tee` transcribes data from standard input to standard output and simultaneously copies this data to the specified file.

OPTIONS

`–a` (append) `tee` appends its output to the original contents of *file* if *file* already exists when `tee` is invoked.

`–a` not specified:

If *file* exists when `tee` is invoked, the original contents of *file* are overwritten.

`–i` (i – ignore) The `SIGINT` signal [see `kill(1)`] is ignored.

`––` End of the list of options. Must be specified if *file* begins with `–`.

file Name of the file to which `tee` is to write its output. If *file* does not exist when you call `tee`, a new file is created; if it does exist, `tee` either overwrites its contents or appends the output to it, depending on whether or not the `–a` option is used. If you specify more than one file, `tee` writes its entire output to each file.

file not specified:

`tee` writes its output to standard output only.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Use of the `tee` command in a pipe:

```
$ (date; who) | tee security | wc –l
      5
$ cat security
Fri Aug 02 14:16:10 MET 1991
felix    tty012   Aug  2 10:37
jane     tty013   Aug  2 08:37
tina     tty003   Aug  2 11:54
eddy     tty004   Aug  2 11:13
```

The `date` and `who` commands send the current date and information on all users currently logged in to standard output. The first pipe redirects this output to `tee`'s standard input. `tee` reads this input and copies it to the file `security` and to standard output. The second pipe sends the standard output of `tee` to the standard input of the `wc –l` command; `wc –l` writes the number of read lines to standard output:

`5` (4 users logged in and 1 line for the date).

User's Reference Manual

test(1)

NAME

`test, [ -` test conditions

SYNOPSIS

`test` *expression*

`[` *expression* `]`

`[[` *expression* `]]`

DESCRIPTION

The utility `test` or the shell built-in `[` is used to check whether specific conditions are satisfied. Such conditions may be:

- file attributes,
- characteristics and comparisons of strings, and
- algebraic comparisons of integers.

Conditions can be negated and several conditions may be combined with one another.

`test` returns the following results:

- Exit status 0 (true) if the condition is satisfied.
- Exit status 1 (false) if the condition is not satisfied or was not fully defined. A false exit status is also returned if you do not specify a condition.

Depending on the exit status, you can execute various commands, terminate loops, etc.

Along with the `/usr/bin/test` command, there are also built-in `test` commands in the shell.

- The following alternative format exists for the built-in `test` command in the Bourne shell `sh`. It has the same effect.

  `[` *expression* `]`

  The square brackets must be specified, as must the blank, before or after *expression*. The shell executes the built-in `sh` command `test` in this case as well.
- The following alternative format exists for the built-in `test` command in the Korn shell `ksh`. It has the same effect.

  `[` *expression* `]` or `[[` *expression* `]]`

  The square brackets must be specified, as must the blank, before or after *expression*. There are additional options for *expression* in the Korn shell [see `ksh(1)` , **CONDITIONAL EXPRESSIONS**].

OPERANDS

expression
: One or more primary expressions which may be combined (see **Combining conditions**).

`test` checks the following conditions:

File attributes

`-r` *file*
: True if *file* exists and you have read permission.

`-w` *file*
: True if *file* exists and you have write permission.

`-x` *file*
: True if *file* exists and you have execute permission.

`-f` *file*

True if *file* exists and is a regular file.

`-d` *file*
: True if *file* exists and is a directory.

`-h` *file*
: True if *file* exists and is a symbolic link. Normally symbolic links are followed by all other conditions.

`-c` *file*
: True if *file* exists and is a character special file.

`-b` *file*
: True if *file* exists and is a block special file.

`-p` *file*
: True if *file* exists and is a named pipe (FIFO).

`-u` *file*
: True if *file* exists and its set-user-id bit is set.

`-g` *file*
: True if *file* exists and its set-group-id bit is set.

`-k` *file*
: True if *file* exists and its sticky bit is set.

`-s` *file*
: True if *file* exists and is not empty.

`-t` *filedescr*
: True if the *filedescr* is associated with a terminal.

filedescr
: The following may be entered for *filedescr*:

`0` for standard input

`1` for standard output

`2` for standard error.

file Name of the file or directory whose attributes are to be tested. Relative or absolute path names may also be specified.

If you use shell metacharacters in the file names, `test` only checks the first file that matches this name.

If you specify the null string for *file*, i.e. a pair of single quotes '' or double quotes "", `test` interprets *file* as the name of your current directory.

If you do not specify *file*, `test` issues an error message and terminates with exit status 1.

A shell parameter may also be specified for *file*. This parameter should always be enclosed in double quotes "...". If the corresponding shell variable has not been defined, the null string will be passed as an argument to `test`. The quotes thus guarantee that `test` is always supplied an argument during parameter substitution.

Characteristics and comparisons of strings

Any arbitrary sequence of characters may be specified as a string. Blanks and tabs included in the string must be escaped. If the string is to be protected from interpretation by the shell, the relevant metacharacters can be escaped by preceding them with a backslash \ or by enclosing the entire string within single or double quotes.

The null string can be specified by a pair of consecutive double quotes or single quotes. If you do not specify a string, `test` issues an error message and terminates with exit status 1.

A shell parameter may also be specified as a string. This parameter should always be enclosed in double quotes. If the corresponding shell variable has not been defined, the null string will be passed as an argument to `test`. The double quotes thus guarantee that `test` is always supplied an argument during parameter substitution.

[–n] *string*

True if the specified *string* is not the null string, i.e. has a non-zero length.

This option allows you to test whether there is a value assigned to a shell variable. The corresponding shell parameter must be enclosed within double quotes.

–n not specified:

The meaning is the same as above, but the condition is easier to read if –n is specified.

–z *string*

True if the specified *string* is the null string, i.e. has a length of zero.

This enables you to test whether there is no value assigned to a shell variable. The corresponding shell parameter must be enclosed within double quotes.

string1 = *string2*

True if the two strings are identical. The blanks before and after the equals sign are mandatory, since test expects this character as an independent argument. If you are comparing shell parameters, they should be enclosed within double quotes.

string1 != *string2*

True if the two specified strings are not identical. The blanks surrounding the != (not equal to) sign are required, since test expects this character as an independent argument. If shell parameters are being compared, they must be quoted.

string

True if *string* is not the null string.

Algebraic comparison of integers

Integers can either be specified directly or as values of shell variables. There is no limit to the size of integer values you can specify, nor to the size of values you can define for a shell variable.

If you specify a shell parameter as a number, you should always enclose it within double quotes "...". This ensures that test receives the null string as an argument if the corresponding shell variable has not been defined. Quoting thus guarantees that an argument will be supplied to test when parameter substitution is performed.

int1 op int2

The two integers *int1* and *int2* are algebraically compared by test on the basis of the operator specified in *op*.

test expects operators as independent arguments; they must therefore be given between two blanks.

op can be any of the following:

–eq True if the two integers are algebraically equal.

–ne True if the two integers are algebraically not equal.

–ge True if *int1* is algebraically greater than or equal to *int2*.

–gt True if *int1* is algebraically greater than *int2*.

–le True if *int1* is algebraically less than or equal to *int2*.

–lt True if *int1* is algebraically less than *int2*.

Negating conditions

! *condition*

True if the specified condition is not satisfied. The exclamation mark must be followed by a blank.

Example:

```
$ ! –r file
```

test returns an exit status of 0 (i.e. true) if you are not permitted to read the specified file.

Combining conditions

Primary conditions can be linked with one another to form a *compound expression*. The condition itself can also be negated.

Since the corresponding relational operators are expected as independent arguments by `test`, they must be enclosed by one blank on either side.

The `find` command searches directories for files that satisfy given conditions. Conditions for `test` are combined in a similar way.

The following operators can be used to combine conditions:

condition `-a` *condition*
: Logical AND operator, i.e true if all conditions combined in this way are satisfied. Each `-a` operator must be preceded and followed by a blank.

condition `-o` *condition*
: Logical OR operator, i.e true if at least one of the conditions is satisfied. Each `-o` operator must be preceded and followed by a blank.

`\(` *compexpr* `\)`
: *compexpr* represents a compound expression comprising two or more arbitrarily grouped conditions. The operator that precedes conditions grouped within parentheses applies to the whole compound expression (i.e. all conditions) and not just to the condition that immediately follows it.

As the shell has its own interpretation of parentheses, they should be escaped with a backslash when used within `test`. Each operator and expression in the compound expression *compexpr* must be separated from one another by blanks.

Example:

```
$  ! \( -r file -o -w file \)
                 |
                OR
```

The expression within parentheses is true if you have read or write permission for the specified file. The negation operator (`!`) negates the whole expression. Thus, if you have neither read nor write permission for the specified file, `test` will return an exit status of zero (true).

Precedence of operators

The order of precedence for operators used with `test` is as follows:

Parentheses over negation over AND over OR.

The conditions within parentheses in the preceding example are thus evaluated first and then negated.

EXIT STATUS

`0` If the specified expression is true

`1` If the specified condition is false, a condition is not fully specified, or no condition has been given.

`>1` Error

ERROR MESSAGES

`test: argument expected`

This error message is issued if you have failed to specify a condition fully, i.e. if a file, string, or number is missing in the specification.

Shell parameters must therefore always be enclosed within double quotes. Otherwise, an argument will be missing whenever the corresponding shell variable is undefined.

EXAMPLES

Example 1:

The following shell script tests whether the specified positional parameter is the name of a file or of a

directory.

```
if test –f "$1"                 # or: if [ –f "$1" ]
   then
   echo $1 is a file
   elif test –d "$1"            # or: elif [ –d "$1" ]
      then
      echo $1 is a directory
fi
```

Since the $1 positional parameter has been quoted, `test` substitutes the current directory for it if you call the shell script without further arguments.

If the quotes were omitted in this case, `test` would terminate with an error message.

Example 2:

The following script makes use of the operator `–gt` to test whether the first file specified in the command line, i.e. `$1`, contains more lines than the one specified after it (`$2`):

```
if [ `wc –l "$1"` –gt `wc –l "$2"` ]
   then echo $1 contains more lines than $2
fi
```

The `wc –l` command counts and outputs the number of lines in each of the two files. The shell substitutes the specification in the backquotes for the appropriate number of lines.

Example 3:

A test is to be performed to check whether a value has been assigned to the shell variable `TAPE`. This can be done in various ways:

(a)
```
if [ ! –z "$TAPE" ]
       then echo a value is assigned to the TAPE variable
       else ....
fi
```

(b)
```
if [ –n "$TAPE" ]
       then echo a value is assigned to the TAPE
       else ....
fi
```

The `–n` is optional and may be dropped.

The shell parameter `$TAPE` is enclosed within double quotes to prevent `test` from issuing an error message in case no value has been assigned to the corresponding variable.

NOTES

`test` exists both as an external command (`/usr/bin/test`) and as a built-in shell command, both in the Korn shell `ksh(1)` as well as in the Bourne shell `sh(1)`. The shell generates a new process to execute `/usr/bin/test`.

Some differences in behavior may occur when using `test`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`find(1)`, `ksh(1)`, `sh(1)`.

User's Reference Manual

test(1-ucb)
BSD Compatibility Package

NAME

`test -` condition evaluation command

SYNOPSIS

`/usr/ucb/test` *expr*

[*expr*]

DESCRIPTION

`test` evaluates the expression *expr* and, if its value is true, it sets a zero (true) exit status; otherwise, a non-zero (false) exit status is set; `test` also sets a non-zero exit status if there are no arguments. When permissions are tested, the effective user ID of the process is used.

All operators, flags, and brackets (brackets used as shown in the second **SYNOPSIS** line) must be separate arguments to the `test` command; normally these items are separated by spaces.

OPTIONS

The following primitives are used to construct *expr*:

`-r` *file*
: True if *file* exists and is readable.

`-w` *file*
: True if *file* exists and is writable.

`-x` *file*
: True if *file* exists and is executable.

`-f` *file*
: True if *file* exists and is a regular file. Alternatively, if `/usr/bin/sh` users specify `/usr/ucb` before `/usr/bin` in their `PATH` environment variable, then `test` will return true if *file* exists and is not a directory. This is also the default for `/usr/bin/csh` users.

`-d` *file*
: True if *file* exists and is a directory.

`-c` *file*
: True if *file* exists and is a character special file.

`-b` *file*
: True if *file* exists and is a block special file.

`-p` *file*
: True if *file* exists and is a named pipe (FIFO).

`-u` *file*
: True if *file* exists and its set-user-ID bit is set.

`-g` *file*
: True if *file* exists and its set-group-ID bit is set.

`-k` *file*
: True if *file* exists and its sticky bit is set.

`-s` *file*
: True if *file* exists and has a size greater than zero.

`-t` [*fildes*]
: true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.

−z *s1*
: True if the length of string *s1* is zero.

−n *s1*
: True if the length of the string *s1* is non-zero.

s1 = *s2*
: True if strings *s1* and *s2* are identical.

s1 != *s2*
: True if strings *s1* and *s2* are not identical.

s1
: True if *s1* is *not* the null string.

n1 −eq *n2*
: True if the integers *n1* and *n2* are algebraically equal. Any of the comparisons −ne, −gt, −ge, −lt, and −le may be used in place of −eq.

−L *file*
: True if file exists and is a symbolic link. With all other primitives, the symbolic links are followed by default.

These primaries may be combined with the following operators:

!
: Unary negation operator.

−a
: Binary *and* operator.

−o
: Binary *or* operator (−a has higher precedence than −o).

(*expr*)
: Parentheses for grouping. Notice also that parentheses are meaningful to the shell and, therefore, must be quoted.

NOTES

The not–a–directory alternative to the −f option is a transition aid for BSD applications and may not be supported in future releases.

The −L option is a migration aid for users of other shells which have similar options and may not be supported in future releases.

If you test a file you own (the −r, −w, or −x tests), but the permission tested does not have the *owner* bit set, a non-zero (false) exit status will be returned even though the file may have the *group* or *other* bit set for that permission. The correct exit status will be set if you are super-user.

The = and != operators have a higher precedence than the −r through −n operators, and = and != always expect arguments; therefore, = and != cannot be used with the −r through −n operators.

If more than one argument follows the −r through −n operators, only the first argument is examined; the others are ignored, unless a −a or a −o is the second argument.

SEE ALSO

find(1), sh(1).

time(1)

NAME

`time` - time a command

SYNOPSIS

`time` [–p] [––] *prog* [*arg* ...]

DESCRIPTION

`time` can be used to measure the execution time of a program or a shell script. After the program or shell script is executed, `time` writes the following times to standard error: `real`, `user`, `sys`.

- `real` is the elapsed time during the invoked process and its child processes, i.e. the time between program call and program termination.
- `user` is the time spent by the process or one of its child processes when executing user code. A process executes user code when it executes machine instructions from its own code segment.
- `sys` is the time spent by the process or one of its child processes when executing system code. A process executes system code when it executes machine instructions from system calls.
- If `time` is called on a multiprocessor system, the sum of user and system time may be greater than real time. A figure of more than 100% for the apparent CPU load is the result of child processes being split between a number of processors.

OPTIONS

No option specified:
: The output has the format *hh*:*mm*:*ss*.*zz*, whereby *hh* stands for hours, *mm* for minutes, *ss* for seconds, and *zz* for hundredths of a second.

–p
: The time is output in seconds as a floating point number with at least one place after the decimal point.

––
: End of the list of options. Must be specified if *prog* begins with –.

prog
: Name of the program (or shell script) to be timed.

arg
: Optional arguments that may be passed to *prog* exactly as if *prog* were called without `time`.

EXIT STATUS

If *command* was called, `time` returns the exit status of the executed command *command*. Otherwise, `time` returns one of the following values:

`1-125`
: An error occurred in the `time` command.

`126`
: The specified *command* exists, but cannot be executed.

`127`
: The specified *command* cannot be found.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Measure the execution time of the `ls` command. The standard output of `ls` is redirected to the file `list`.

```
$ time ls –l >list
```

```
real    0m0.21s
user    0m0.06s
sys     0m0.06s
```

SEE ALSO

times(1), timex(1), xpg4_sh(1), times(2).

User's Reference Manual

times(1)

NAME

`times` - print accumulated times for processes run from the shell

SYNOPSIS

`times`

DESCRIPTION

The shell built-in `times` indicates the accumulated time spent by processes run from the current shell. Times are reported in minutes (`m`) and seconds (`s`) and are listed in terms of user time and system time.

User time is the time processes have spent in user mode; system is the time processes have spent in system mode.

To time a specific command you should use the `time(1)` command.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Finding out the accumulated user and system times of all processes run from the current shell:

```
$ times
0m15s 0m24s
```

Processes have spent 15 seconds in user mode and 24 seconds in system mode.

NOTES

Some differences in behavior may occur when using `times`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh(1)`, `sh(1)`, `time(1)`, `timex(1)`, `times(2)`.

timex(1)

NAME

`timex` - time a command; report process data and system activity(time execution)

SYNOPSIS

`timex` [*option ...*] *cmd* [*arg ...*]

DESCRIPTION

`timex` has a program or shell script executed and reports the time spent in execution. As with `time(1)`, the measured CPU times are output to standard error in the format *hh*:*mm*:*ss*.*zz*, where *hh* are hours, *mm* are minutes, *ss* are seconds and *zz* are hundredths of seconds.

`timex` reports values for `real`, `user`, and `sys`.

- `real` is the elapsed time during the invoked process and its child processes, i.e. the time between program call and program termination.
- `user` is the time spent by the process or one of its child processes when executing user code. A process executes user code when it executes machine instructions from its own code segment.
- `sys` is the time spent by the process or one of its child processes when executing system code. A process executes system code when it executes machine instructions from system calls.

Examples of `timex` output:

```
real         0.04
user         0.01
sys          0.02
```

For longer runtimes:

```
real    22:26.30
user  6:08:53.68
sys      6:47.91
```

You can use options to obtain additional process data on the executed command and its child processes, as well as data on total system activity during command runtime.

Note:

If `timex` is called on a multiprocessor system, the sum of user and system time will be considerably greater than real time. A figure of more than 100% for the apparent CPU load is the result of child processes being split between a number of processors.

OPTIONS

No option specified:

`timex` executes the given command *cmd* and reports the real, user and system time spent in execution of the command.

`-p` (process) `timex` lists process data on *cmd* and its child processes. This option only works if process accounting has been activated.

Options `f`, `h`, `k`, `m`, `r` and `t` are suboptions of `-p` and are only effective when used in conjunction with `-p`.

`-o` `timex` reports the total number of blocks read or written by *cmd* and its child processes. The number of characters transferred is also reported.

This option only works if process accounting has been activated.

`-s` `timex` reports total system activity occurring during the runtime of *cmd*. The report covers the same activities as the `sar` command.

The following options are only effective when used in conjunction with `-p`:

`-f` The output contains additional columns for the `fork` and `exec` flags, and for the system exit status.

`-h` (hog factor) Instead of mean memory size for the process, `timex` reports the fraction of total available CPU time consumed by the process during its execution. This value is known as the "hog factor" and is calculated as: total CPU time divided by elapsed time.

`-k` `timex` reports the cumulative memory usage per minute of process execution.

`-m` (mean core size) `timex` reports the mean core size.

This option is set by default.

`-r` `timex` reports the CPU factor (*user-time*/(*system-time* + *user-time*)).

`-t` *timex* reports system CPU time separately from user CPU time. The number of blocks read or written and the number of characters transferred are always reported.

cmd Program or shell script executed by `timex`.

arg Argument for the program or shell script *cmd*. The number of arguments required depends on the *cmd* involved.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The time required to execute a `find` command is to be measured and reported. To do this, you enter `timex` followed by the full syntax of the `find` command with all the necessary arguments:

```
$ timex find . -name test -print
real        2.32
user        0.10
sys         1.65
```

SEE ALSO

`sar(1)`, `time(1)`, `times(2)`.

touch(1)

NAME

`touch -` update access and modification times of a file

SYNOPSIS

Format 1

`touch` [`–acm`] {`–r` *reference_file* | `–t` *time*} [`––`] *file ...*

Format 2

`touch` [`–acm`] [*time*] [`––`] *file ...*

DESCRIPTION

`touch` is used to set the access and modification times of files to the current or specified date.

OPTIONS

Format 1: New synopsis for touch

No option specified:

`touch` sets the time of the last modification and the time of the last access for the specified files to the current date. If a file does not yet exist `touch` creates one. Specifying `touch` without an option has the same effect as `touch –am`.

`–a` (a – access time) `touch` sets the time of the last access for the specified files.

Neither `–a` nor `–m` specified: `touch` sets the time of the last modification and the time of the last access.

`–c` `touch` does not create files that do not exist. No message is output here; the exit status is not set to an error.

`–m` (m – modification time) `touch` sets the time of the last modification for the specified files.

Neither `–a` nor `–m` specified: `touch` sets the time of the last modification and the time of the last access.

`–r` *reference_file*

Modification and access time of the file *reference_file* are used instead of the current date.

`–r` must not be combined with `–t` or the obsolete argument *time*.

`–t` *time*

The *time* specified here is used instead of the current date. *time* is specified in the following format:

[[*CC*]*YY*]*MMDDhhmm*[**.***SS*]

CC The first two digits of a year: `19` or `20`

CC not specified:

If the two-digit year notation *YY* is in the range `70` to `99`, the 20th century is assumed; if it is in the range `00` to `37`, the 21st century is assumed. All other entries are rejected.

YY Two-digit year notation

Only the digits >`69` and <`38` may be entered. If the year number specified is <`38`, this is beyond the year 2000.

YY not specified: The current year is assumed.

MM Two-digit month notation (01 through 12)

DD Two-digit day notation (01 through 31)

hh Two-digit hour notation (00 through 23)

mm Two-digit minute notation (00 through 59)

SS Two-digit second notation (00 through 61)

The values `60` and `61` are intended for leap seconds.

SS not specified: The value 0 seconds is assumed.

`-t` must not be combined with `-r` or the obsolete argument *time*.

`--` End of the list of options. Must be specified if *file* begins with `-`.

file Name of the input file. `touch` processes all types of files and directories. Several filenames can be specified for each call.

Format 2: Old synopsis for touch

time not specified:

`touch` sets the access and modification times of the named files to the specified or current date. If a file does not exist, it is created. Calling `touch` without options is thus equivalent to `touch -am`.

`-a` Updates only the access time of the named files to the specified or current date.

If neither `-a` nor `-m` is specified: `touch` sets the access and modification times of the named files to the specified or current date.

`-c` Prevents `touch` from creating a file if the named file did not previously exist. No corresponding message is issued.

`-m` Updates only the modification time of the named files to the specified or current date.

If neither `-a` nor `-m` is specified: `touch` sets both the access and modification times of the named files to the specified or current date.

mmddhhmm[*yy*]

Date and time to which the access and/or modification times of the specified files are to be set. The date/time argument comprises eight to ten digits with the following significance:

month (*mm*); day (*dd*); hour (*hh*); minute (*mm*); year (*yy*).

yy not specified: `touch` assumes you mean the current year.

Note: Time notations in this format are obsolete. Filenames consisting only of digits can cause problems, as `touch` may be interpreted as a date notation. You should replace *time* with the `-t` option.

`--` End of the list of options. Must be specified if *time* was not specified and *file* begins with `-`.

file Name of the input file. `touch` processes all types of files, including directories. Several file names may be specified in one call.

ERROR MESSAGES

`date: bad conversion`

You have specified an illegal date, e.g. 13010000.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

`TZ` defines the time zone for the `-t` option.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Set the access and modification times of `file` to the current date. No new file is to be created if this file does not exist.

```
$ touch -c file
```

The `ls –l` command can be used to display the time of the last modification; `ls –lu` outputs the access time.

Example 2

Set the access time of `file` to 9 o'clock on 8/26:

```
$ touch –a –t 08260900 file
$ ls –lu file
-rw-r--r--   1 bert    ap343       736  Aug 26 09:00 file
```

SEE ALSO

`date`(1), `ls`(1), `utime`(2).

User's Reference Manual

tput(1)

NAME

tput - initialize a terminal or query the terminfo database

SYNOPSIS

Format 1

tput [–T *type*] *capname* [*parameter* ...]

Format 2

tput [–T *type*] –S <<!
> *capname* [*parameter*] ...
> !

Format 3

tput [–T *type*] init

Format 4

tput [–T *type*] reset

Format 5

tput [–T *type*] longname

DESCRIPTION

You can use tput to

- indicate one capability of a terminal (**Format 1**)
- list a number of capabilities of a terminal (**Format 2**)
- initialize a terminal (**Format 3**)
- reset a terminal (**Format 4**)
- display the long name of a terminal (**Format 5**)

OPTIONS

Format 1:Indicating one capability of a terminal

tput [–T *type*] *capname* [*parameter* ...]

tput indicates capabilities of a terminal as defined in the terminfo database [see terminfo(4)].

–T *type*

type indicates the type of terminal whose capabilities are to be queried. If the –T *type* option is specified, the shell variables LINES and COLUMNS and the layer size will not be referenced.

–T *type* not specified:

type defaults to the value of the environment variable TERM.

capname

capname is the alias (abbreviated form) of a terminal capability as defined in the terminfo source file. To find out whether your terminal has a particular capability (attribute), you specify the appropriate alias *capname* [see terminfo(4)]. Depending on the type of attribute, tput responds as follows:

- Boolean type attribute: tput simply sets an exit status (0 for true is the terminal has the attribute, 1 for false if it does not).

  In the Bourne and Korn shells you can inspect the exit status with echo $?.
- string type attribute: tput returns the appropriate string.
- integer type attribute: tput returns an integer.

If *capname* is an attribute for which no value is assigned to the specified terminal type in the `terminfo` database, `tput` returns –1.

parameter

If the *capname* attribute is a string that takes parameters, you can enter the parameters in *parameter*. *capname* and *parameter* are passed to `tput` as a compound string. An all numeric parameter is passed as a number.

Format 2: Listing a number of capabilities of a terminal

```
tput [–T type] –S <<!
>   capname [parameter]
 ...
> !
```

`tput` can be used to list multiple capabilities of the user's terminal. You pass the capabilities from standard input rather than from the command line (see *Example 4*). Only one terminal capability is permitted per line. If the –T *type* option is specified, the shell variables `LINES` and `COLUMNS` will not be referenced. The meanings of the exit statuses 0 and 1 change (see **ERROR MESSAGES**).

–T *type*

type indicates the type of terminal whose capabilities are to be queried. If the –T *type* option is specified, the shell variables `LINES` and `COLUMNS` and the layer size generated with the `layers` command remain unchanged.

–T *type* not specified:

type defaults to the value of the `TERM` environment variable.

capname

capname is the alias (abbreviated form) of a terminal capability as defined in the `terminfo` source file. To find out whether your terminal has a particular capability (attribute), you specify the appropriate alias *capname* [see `terminfo(4)`]. Depending on the type of attribute, `tput` responds as follows:

- Boolean type attribute: `tput` simply sets an exit status (0 for `true` is the terminal has the attribute, 1 for `false` if it does (not). In the Bourne and Korn shells you can inspect the exit status with `echo $?`.
- string type attribute: `tput` returns the appropriate string.
- integer type attribute: `tput` returns an integer.

If *capname* is an attribute for which no value is assigned to the specified terminal type in the `terminfo` database, `tput` returns –1.

Note: Using the command `infocmp –I term` [see `infocmp(1M)` and `term(4)`] you can obtain a full description of your terminal. `term` is the name of your terminal as listed in the output of the command `ls –C /usr/share/lib/terminfo/?`.

parameter

If the *capname* attribute is a string that takes parameters, you can enter the parameters in *parameter*. *capname* and *parameter* are passed to `tput` as a compound string. An all numeric parameter is passed as a number.

Format 3: Initializing a terminal

`tput [–T type] init`

If the `terminfo` database contains an entry for the user's terminal, `tput` initializes the terminal in accordance with the terminal type specified in –T *type*. `tput` performs the following individual activities:

- If present, the terminal's initialization strings are output (*capname* `is1`, `is2`, `is3`, `if`, `ifprog`).
- Any delays (e.g. newline) specified in the entry are set in the terminal driver.
- Tabs expansion is turned on or off according to the specification in the entry.
- If tabs are not expanded, the standard tabs are set (every 8 spaces).

If the terminfo entry does not contain the information needed for any of these activities, that activity is silently skipped.

–T *type*

This indicates the type of terminal whose capabilities you want to use to initialize your terminal.

–T *type* not specified:

type defaults to the value of the environment variable TERM.

Format 4: Resetting a terminal

tput [–T *type*] reset

tput outputs the terminal's reset strings (*capname* rs1, rs2, rs2, rf). If the reset strings are not present but initialization strings are, the initialization strings are output. Otherwise, reset behaves identically to init.

–T *type*

This indicates the type of terminal whose capabilities you want to use to reset your terminal.

–T *type* not specified:

type defaults to the value of environment variable TERM.

Format 5: Displaying the long name of a terminal

tput [–T *type*] longname

If the terminfo database is present and contains an entry for the user's terminal (see **Format 1**, –T *type* option), the long name of the terminal is output. This name is the last name in the first line of the description of the terminal in the terminfo database [see terminfo(4)].

EXIT STATUS

When *capname* is of type boolean and the –S option is not set:

0 if the specified terminal type has the capability

1 if the specified terminal type does not have the capability

When *capname* is of type string and the –S option is not set:

0 if the *capname* capability is defined for the terminal type

1 if the *capname* capability is not defined for the terminal type (nothing is written to standard output)

When *capname* is of type boolean or string and the –S option is set:

0 if all lines could be processed successfully

1 exit status 1 can never occur as no indication of which line failed can be given

When *capname* is of type integer:

0 The exit status is always 0. You can test the value of the standard output to determine whether *capname* is defined for the specified terminal type or not. –1 on standard output indicates that *capname* is not defined.

Error situations are indicated by the exit status >1:

2 Application error

3 There is no information available for the specified terminal.

4 One of the specified arguments is invalid.

>4 Another error has occurred.

ERROR MESSAGES

Depending on the exit status, tput issues one of the following error messages:

tput: unknown terminal *type*

Terminal type *type* unknown or no terminfo database present, exit status 3.

tput: unknown terminfo capability *capname*

Self-explanatory, exit status 4.

ENVIRONMENT VARIABLES

`TERM` Default value used for the type of terminal when `-T` *type*

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Initializing a terminal

```
$ tput init
```

A sequence of control characters that initializes the screen is passed to the user's terminal. The sequence used is the one appropriate to the terminal type defined in the `TERM` environment variable.

Example 2

Displaying the number of columns for the current terminal

```
$ tput cols
80
```

Example 3

Indicating whether the current terminal is a hardcopy terminal

```
$ tput hc
$ echo $?
1
```

Since `hc` is a Boolean type value, `tput` simply returns an exit status, which you can inspect with `echo $?`. The returned value of 1 indicates that the terminal is not a hardcopy terminal.

Example 4

Displaying several capabilities in one `tput` invocation

```
$ tput -S <<!
> clear
> cup 10 10
> bold
> !
```

The screen is cleared, the cursor moved to position 10,10 and bold (extra bright) mode turned on. The list is terminated by an exclamation mark in a separate line.

FILES

`/usr/share/lib/terminfo/?/*`
: Directory containing binary versions of the terminal type descriptions

`/usr/include/curses.h`
: `curses(3X)` header file

`/usr/include/term.h`
: `terminfo(4)` header file

`/usr/lib/tabset/*`
: Information on tab handling (see `terminfo(4)` , section "Tabs and initialization")

SEE ALSO

clear(1), stty(1), tabs(1), infocmp(1M), profile(4), term(4), terminfo(4), term(5), term_names(5).

User's Reference Manual

tr(1)

NAME

`tr -` transliterate characters

SYNOPSIS

`tr [ –c ] [ –d ] [ –s ] [ –– ]` *string1* [*string2*]

DESCRIPTION

`tr` reads an input text from standard input, replaces or deletes selected characters from it, and writes the result to standard output.

Character substitution

If you wish to substitute characters using `tr`, you must define both strings. The `–d` option must not be specified.

`tr` replaces each character in the input text that appears in *string1* with the appropriate character from *string2*: the *i*-th character in *string1* is replaced in the input text by the *i*-th character in *string2*. If *string2* is shorter than *string1*, the characters from *string1*, for which there are no corresponding characters in *string2*, are not replaced (see *Example 1*).

Character deletion

`tr` deletes characters if the `–d` option (any character) or the `–s` option (only characters that appear a number of times) is specified.

`–d` considers all characters from *string1*. `–s` works with the first or second string, depending on whether the `–d` option is specified as well.

OPTIONS

`–c` (c – complement) Complements *string1* with respect to the currently applicable character set (octal values 001 through 377). The complemented *string1* then then contains all characters of the currently applicable character set except for those specified in the original *string1*.

`–c` is always executed first by `tr`. In the input text, `tr` then replaces the *i*-th character in the complemented *string1* with the *i*-th character in *string2*, or deletes the characters that appear in the complemented string.

`–d` (d – delete) All input characters that appear in *string1* are deleted.

Together with the `–c` option: All input characters that appear in the complemented string *string1* are deleted.

Together with the `–s` option: After deletion, each sequence of the same characters that appear in *string2* are truncated to a single character in the output text. If `–s` is not specified, *string2* is ignored. If *string2* is not specified, `–s` is ignored.

`–s` (s – squeeze) `tr` truncates each sequence of the same characters that appear in *string1* or *string2* to a single character in the output text (see *Example 3*). *string2* is then only used by `–s` if the `–d` option is specified as well.

string1 [*string2*]

In *string1*, specify the characters to be replaced or deleted. *string2* is the replacement string (except where `–d` and `–s` are combined).

The characters in both strings must be specified without intervening blanks or other delimiters.

If a string contains metacharacters that have a special meaning for the shell, these metacharacters must be escaped by enclosing the entire string in single quotes '...' or by preceding each such character with a backslash \.

The strings can contain the following specifications:

character
: Any printable *character*.

octal_number
: Whereby *octal_number* is a one-, two-, or three-digit octal number. The backslash must be escaped so that the digits are recognized as an octal number.

`tr` also processes the NUL character (\000).

Warning: In previous versions, the NUL character was always deleted as an input character.

metacharacter
: As escape sequences (same as for the `printf` command). The following escape sequences can be specified:

`\\` Backslash (for distinguishing octal numbers)

`\a` Warning, alert

`\b` Backspace

`\f` Form feed

`\n` Newline

`\r` Carriage return

`\t` Tab

`\v` Vertical tab

`a-z` or `[a-z]`
: Stands for the set of characters from `a` to `z` inclusive. [Characters are sorted in the currently applicable collating sequence. Unlike in internationalized regular expressions, a and z must be ordinary characters, i.e. not equivalence class expressions `[=`*c*`=]` or collating symbols `[.`*cc*`.]`.

`[:`*class*`:]`
: *class* specifies a character class, similar to internationalized regular expressions. The following values are possible for *class*:

`alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`.

Character classes must not be specified in a replacement string. Exception: The classes `lower` and `upper` are permitted if the corresponding character class is specified on the same side in *string1*.

`[=`*equivalence*`=]`
: *equivalence* specifies an equivalence class, similar to internationalized regular expressions.

Equivalence classes must not be specified in a replacement string.

`[a*`*n*`]`
: Stands for *n* repetitions of the `a`, e.g. `[a*3]` stands for `aaa`. Only useful in a replacement string.

If the first digit of *n* is 0, *n* is considered octal; otherwise, it is taken to be decimal.

If *n* is 0 or is omitted, it is taken to be "huge", meaning that the preceding character is to be repeated as often as required to pad *string2* to the length of *string1* (see *Example 1*).

string2 not specified: *string1* (possibly complemented, see `-c`) is used for *string2*.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed.

In expressions in square brackets, the `LC_COLLATE` environment variable governs the scope of character ranges (e.g. `[a-z]`). `LC_CTYPE` also specifies which characters are included in the currently valid character set in conjunction with the `-c` option.

If `LC_MESSAGES`, `LC_COLLATE` or `LC_CTYPE` is undefined or is defined as the null string, it defaults to the

value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

tr without options: simple examples to demonstrate how tr works:

```
$ cat days
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
$ tr TS ts <days
Monday tuesday Wednesday thursday Friday saturday sunday
```

Here tr replaces all occurrences of T with t and of S with s.

```
$ tr TSF ts <days
Monday tuesday Wednesday thursday Friday saturday sunday
```

The second string has fewer characters than the first in this case. tr replaces T by t and S by s, but leaves F unaltered.

Now let us try replacing all lowercase letters with x. The following solution is **not** suitable:

```
$ tr '[a-z]' x <days
Mondxy Tuesdxy Wednesdxy Thursdxy Fridxy Sxturdxy Sundxy
```

tr has clearly only replaced occurrences of a with x. To replace all lowercase letters, we need to call tr as follows:

```
$ tr '[a-z]' '[x*]' <days
Mxxxxx Txxxxxx Wxxxxxxxx Txxxxxxx Fxxxxx Sxxxxxxx Sxxxxx
```

Each character in *string1* now has a corresponding x in *string2*, as the asterisk (*) causes *string2* to be padded with x as often as required. The single quotes are essential, since the strings include shell metacharacters.

Example 2

We want to create a list of all the words that appear in textfile, one word to a line, a word being defined as any consecutive string consisting only of letters.

```
$ cat textfile
"When shall we three meet again?
In thunder, lightning, or in rain?"
$ tr -cs '[:alpha:]' '[\n*]' <textfile
When
shall
we
three
meet
again
In
thunder
lightning
or
in
rain
```

Example 3

Deleting a non-printing character from a file (tr –d):

```
$ tr –d '\016' <file
```

tr deletes from the file the character with the octal code 016 and displays the result on standard output.

SEE ALSO

ed(1), sed(1), sh(1).

User's Reference Manual

tr(1-ucb)
BSD Compatibility Package

NAME

`tr -` translate characters

SYNOPSIS

`/usr/ucb/tr` [`–cds`] [*string1* [*string2*]]

DESCRIPTION

`tr` copies standard input to standard output with substitution or deletion of selected characters. The arguments *string1* and *string2* are considered sets of characters. Any input character found in *string1* is mapped into the character in the corresponding position within *string2*. When *string2* is short, it is padded to the length of *string1* by duplicating its last character.

In either string the notation:

a–b

denotes a range of characters from *a* to *b* in increasing ASCII order. The character `\`, followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. As with the shell, the escape character `\`, followed by any other character, escapes any special meaning for that character.

OPTIONS

Any combination of the options `–c`, `–d`, or `–s` may be used:

`–c` Complement the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal.

`–d` Delete all input characters in *string1*.

`–s` Squeeze all strings of repeated output characters that are in *string2* to single characters.

EXAMPLE

The following example creates a list of all the words in *filename1* one per line in *filename2*, where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect `\` from the shell. 012 is the ASCII code for NEWLINE.

`tr –cs A–Za–z '\012' <`*filename1* `>`*filename2*

NOTES

Will not handle ASCII NUL in *string1* or *string2*. `tr` always deletes NUL from input.

SEE ALSO

`ed(1)`, `ascii(5)`.

User's Reference Manual

trap(1)

NAME

`trap -` define signal handling

SYNOPSIS

Format 1

`trap` [*command_list*] *signal_number ...*

Format 2

`trap`

DESCRIPTION

The shell built-in `trap` is used to define how the current shell is to react to subsequent signals received. You can thus use `trap` in shell scripts to have specific "clean-up" tasks performed before the script terminates, or to define positions at which no interrupts are to take place. `trap` has two functions:

- `trap` defines how the shell is to react to a signal (**Format 1**):
  - The shell executes the commands specified in the `trap` command line.

    Any command possibly terminated earlier will not be resumed after the command list is executed. Shell scripts are resumed with the command that follows the one that was interrupted.
  - The shell ignores the specified signal.
  - The shell reverts to default actions for the specified signals. `trap` can be used to reset signal handling to the default behavior.
- `trap` displays all signals for which signal handling defaults were modified in the current shell (**Format 2**).

Format 1: Define signal handling

`trap` [*command_list*] *signal_number ...*

command_list

determines how the shell is to react to the signals specified, i.e. whether it is to

- execute commands if the signal specified is received
- ignore the specified signal
- revert to default actions for the signal specified.

Executing commands

You can specify one or more commands for *command_list* (e.g. `trap pwd 2`). These commands are to be executed if the signal specified is received. If you specify more than one command, you should separate these by semicolons. The semicolon must be quoted to prevent immediate interpretation by the shell.

The command list must be a single argument, so it must always be enclosed in single quotes '...'or double quotes "..." if argument separators or semicolons appear in it.

If the given *command_list* is not a null string, the signal handling behavior it defines will only be valid in the current shell. `trap` must be explicitly called in each subshell; otherwise, default actions are automatically restored.

It is relevant to note here that the specified commands are interpreted twice by the shell:

- once when the shell sets the `trap`, and
- once when the corresponding signal occurs, and the shell invokes `trap` to execute the defined commands.

It can thus make a difference whether single or double quotes are used:

'*command_list*'

Special characters are not interpreted until the shell actually invokes `trap` and executes the commands. Shell variables are thus evaluated only when the `trap` routines are executed.

"*command_list*"

The characters `$`, `\`, and `` `...` `` are interpreted by the shell when the `trap` is set. Shell variables are often still undefined at this stage, however.

If you want to prevent the shell script from being executed further when a signal has been received, specify `exit` as the last command in the command list.

Ignoring signals

If a null string ("" or '') is specified for *command_list* the specified signals are ignored (e.g. `trap "" 2`).

The corresponding signals are also ignored in every subshell.

Resetting signal handling to default

If *command_list* is not specified (e.g. `trap 2`), the shell reverts to default actions for the signals specified.

signal_number

Number identifying the signal for which the shell is to execute the defined actions [see `signal(5)`]. Several blank-separated signals may be specified. *command_list* will then be executed as soon as any one of these signals is received.

In the Korn shell `ksh` you can specify the signal names instead of the signal numbers.

The following signals are relevant for the shell:

0	EXIT
1	SIGHUP
2	SIGINT
3	SIGQUIT
6	SIGABRT
9	SIGKILL
14	SIGALRM
15	SIGTERM

If `0` is specified for *signal_number* this causes the specified *command_list* to be executed on exiting the current shell; 0 is not a signal. This means

- if you have called `trap` interactively, *command_list* will be executed as soon as you press the CTRL-D key.
- if `trap` is specified in a shell script, *command_list* will be executed after this script.

Signal `9` (`SIGKILL`) always kills the process, so `trap '' 9` will not work.

Signal `11` (`SIGSEGV`) cannot be specified with `trap`.

Format 2: Display changed signal handling routines

`trap` The `trap` command with no arguments prints the list of commands associated with each signal number for which signal handling has been altered in the current shell. The list is written to standard output in the following format:

signal_number: *command_list*

or in the Korn shell `ksh`:

`trap` — *command_list signal*

Only the signal numbers for which default actions were modified earlier (with `trap`) are displayed.

Signal handling in the shell

A process can receive a signal at any time during its execution. Such signals may either be generated by the process itself, by another process or by the user at the terminal (e.g. by pressing DEL). The shell then reacts to the signal in one of the following ways:

- It ignores the incoming signal.
- It terminates.
- It calls a function to deal with the signal.

The following signals are relevant for the shell:

Signal number	Signal	Cause
1	SIGHUP	Disconnection of link to terminal
2	SIGINT	DEL
3	SIGQUIT	CTRL-\
9	SIGKILL	The command `kill –9` *PID*, where *PID* is the process identification number of the corresponding shell
15	SIGTERM	The command `kill –15` *PID*, where *PID* is the process identification number of the corresponding shell

Depending on whether or not you have defined signal handling with `trap`, the shell will react to these signals as follows:

Signal number	Interactive shell	Shell script in the background
1	ignore	abort
2	take defined action after next command or after RETURN □ otherwise: ignore	ignore
3	take defined action after next command or after RETURN □ otherwise: ignore	ignore
9	abort	abort
15	ignore	ignore

Depending on whether or not you have defined signal handling with `trap`, a shell script or the current foreground process in a shell script will react as follows (where *PID* is the process ID of the shell script):

Signal handling undefined for shell script and current foreground process in shell script

Signal number or key	Shell script	Foreground process in shell script
`kill –1` *PID*	abort immediately	continue running
`kill –2` *PID*	abort after normal termination of foreground process	continue running
DEL	abort immediately	abort immediately
`kill –3` *PID*	abort after normal termination of foreground process	continue running

`CTRL-\`	abort immediately	abort immediately, writing core dump to disk
`kill –9` *PID*	abort immediately	continue running
`kill –15` *PID*	abort after normal termination of foreground process	continue running

Signal handling undefined for shell script, defined for current foreground process in shell script

Signal number or key	Shell script	Foreground process in shell script
`kill –1` *PID*	abort immediately	continue running
`kill –2` *PID*	abort after normal termination of foreground process	take defined action
`DEL`	abort after normal termination of foreground process	take defined action
`kill –3` *PID*	abort after normal termination of foreground process	take defined action
`CTRL-\`	abort after normal termination of foreground process	take defined action
`kill –9` *PID*	abort immediately	continue running
`kill –15` *PID*	abort after normal termination of foreground process	take defined action

Signal handling defined for shell script, undefined for current foreground process in shell script

Signal number or key	Shell script	Foreground process in shell script
`kill –1` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)
`kill –2` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)
`DEL`	take defined action; then resume execution	abort immediately
`kill –3` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)
`CTRL-\`	take defined action; then resume execution	abort immediately, writing core dump to disk
`kill –9` *PID*	abort immediately	continue running
`kill –15` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)

Signal handling defined for shell script and current foreground process in shell script

Signal number or key	Shell script	Foreground process in shell script
`kill –1` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)

`kill –2` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)
`DEL`	take defined action; then resume execution	take defined action; then resume execution
`kill –3` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)
`CTRL-\`	take defined action; then resume execution	take defined action; then resume execution
`kill –9` *PID*	abort immediately	continue running
`kill –15` *PID*	take defined action after normal termination of foreground process; then resume execution	continue running (signal not delivered)

The `signal(2)` function can be used in C programs to define how signal handling is to be implemented [see `signal(2)`].

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

EXAMPLES

Example 1

To ensure that signal 2 is ignored in a shell script, the following line is included in it:

```
trap ''2
```

The two single quotes, i.e. the null string, cause signal 2 to be ignored. This means that the shell script cannot be terminated externally by the DEL key or by the command `kill –2` *process_id*.

Example 2

Use of the command `trap` in an interactive shell:

```
$ trap 'echo Last logged out: `date` >>$HOME/logfile' 0
$ trap
0, echo Last logged out: `date` >>$HOME/logfile
$ CTRL-D
 .
 .
 .
login: rose
Password:
$ cat logfile
Last logged out: Mon Mar 20 22:17:23 MEZ 1989
```

The `trap` in the above example defines that a message is to be written to the file `$HOME/logfile` on exiting the current shell. The command list must be enclosed in single quotes here, since `date` is only to be executed when the shell terminates.

Example 3

The shell script `traptest` illustrates how temporary files can be deleted if signals are received during execution. The script contains the following lines (without the line numbers):

```
1  TMP=/usr/rtmp/$$
2  trap "rm –f $TMP; trap 0; exit 1" 1 2 3 15
3  trap "rm –f $TMP; exit 0" 0
4  ls > $TMP
```

```
.
.
.
```

Line 1:

The file name `/usr/rtmp/$$` is assigned to the variable `TMP`. The shell substitutes the process ID of the current shell for `$$`, thus creating a unique file name.

Line 2:

The command list is enclosed in double quotes, since the `TMP` variable has already been assigned a value. The following actions are defined for signals 1, 2, 3, and 15: delete the file `/usr/rtmp/$$`, reset the end of script definition (0) (see line 3), and terminate the script with exit status1.

Line 3:

0 is the only signal-number specified, i.e. the definition for the end of the script is: delete the file `/usr/rtmp/$$`, and terminate the script with exit status 0. This definition must be reset in line 2 with `trap 0`, since the command `exit` terminates the shell script. Otherwise, line 3 would cause the script to terminate with exit status 0.

Line 4:

The file `/usr/rtmp/$$` is created. This line must not precede the `trap` command; otherwise, if the script terminates before the shell executes the first `trap` command, the file will not be deleted.

In this case the command `exit` should be explicitly invoked at the end of the command list, otherwise the commands that follow `trap` in the script could execute in an undefined state.

NOTES

Some differences in behavior may occur when using `trap`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`exit(1)`, `kill(1)`, `ksh(1)`, `sh(1)`, `signal(5)`.

User's Reference Manual

true(1)

NAME

`true` - return zero exit status

SYNOPSIS

```
true
```

DESCRIPTION

The `true` command simply returns a zero exit status and does nothing else.

You can use `true` in shell scripts to generate the condition "true".

You can use the companion command `false(1)` to generate the condition "false" (non-zero exit status).

EXIT STATUS

```
0
```

EXAMPLES

The following shell script initiates an endless loop which you can terminate by pressing <DEL>.

```
while true
do
.
.
.
done
```

SEE ALSO

`colon(1)`, `false(1)`, `ksh(1)`, `sh(1)`.

User's Reference Manual

truss(1)

NAME

`truss -` trace system calls and signals

SYNOPSIS

`truss` [*option ...*] *command*

DESCRIPTION

`truss` executes the specified command and produces a trace of the system calls it performs, the signals it receives, and the machine faults it causes. Each line of the trace output reports either the fault or the signal name or the system call name with its arguments and return values. System call arguments are displayed symbolically when possible using defines from relevant header files. Error returns are reported using the error names described in `intro_prm2(2)` .

OPTIONS

No option specified:

`truss` writes to standard error a trace of the system calls the specified command performs, the signals it receives, and the machine errors it causes.

With the following options, it is possible to specify the same option more than once with different option arguments. In the event of conflicts, the options to the right override those further to the left.

`-a` `truss` outputs the argument strings which are passed in each `exec(2)` system call.

`-c` `truss` counts the traced system calls, faults and signals rather than displaying the trace, outputting the result after the specified command has terminated or when `truss` is interrupted. The count contains a separate total for system calls, faults and signals.

If `-f` is also specified, the counts also include system calls, faults and signals for all child processes.

`-e` `truss` outputs the environment strings which are passed in each `exec(2)` system call.

`-f` `truss` also traces all the child processes created by `fork(2)` and includes their system calls, faults and signals. The process ID (PID) is included with each line of trace output to indicate which process executed the system call or received the signal.

`-f` not specified:

Only the first-level command or process is traced.

`-i` `truss` does not trace interruptible sleeping system calls, such as `open(2)` and `read(2)` on terminal devices or pipes, until they have terminated.

`-i` not specified:

`truss` traces interruptible sleeping processes once when they have been asleep for more than a second, and again when they have terminated.

`-m` [`!`]*fault*, ...

You specify one or more machine faults that you want `truss` to trace. Faults may be specified either name or number (see `<sys/fault.h>`). If you want all faults traced, you can enter `all` for *fault*. If you put an exclamation mark `!` before *fault*, `truss` excludes the listed faults from the trace output.

`-m` [`!`]*fault* not specified:

has the same effect as `-m all -m !fltpage`, i.e. all machine faults apart from `FLTPAGE` (recoverable page fault) are traced.

`-o` *outfile*

`truss` writes the trace to *outfile*.

`-o` *outfile* not specified:

truss writes the trace to standard error.

-p truss interprets *command* as a list of process IDs (PIDs) for existing processes [see ps(1)] rather than as a command to be executed. truss takes control of each process and begins tracing it provided the user ID (UID) and group ID (GID) [see ps(1)] of the user match those of the process, or the user is the system administrator. Processes can also be specified by their name in the /proc directory, e.g. /proc/1234. This works for remotely mounted proc directories as well.

-r [!]*fd*, ...
truss shows the full contents of the I/O buffer for each read(2) system call on any of the specified file descriptors *fd*. Specifying all for *fd* gives you all file descriptors. If you put an exclamation mark ! before *fd*, truss excludes the listed file descriptors from the trace output. The output is formatted 32 bytes per line and shows each byte as an ASCII character preceded by one blank or as a two-character C language escape sequence for control characters, such as \t for horizontal tab or \n for newline. If ASCII interpretation is not possible, the byte is shown in hexadecimal code.

-r [!]*fd* not specified:

has the same effect as -r !all, i.e. truss does not show the full contents of the I/O buffer for any file descriptor. Only the first twelve bytes of the I/O buffer are show for each traced read(2) or write(2) .

-s [!]*signal*, ...
You specify a comma-separated list of signals that truss is to trace. To trace all signals, you specify all for *signal*. If you put an exclamation mark ! before *signal*, truss excludes the listed signals from the trace output. truss traces the receipt of each specified signal even if the signal is being ignored (not blocked) by the process (blocked signals are not received until the process releases them). Signals may be specified by name or number [see signal(5)].

-s [!]*signal* not specified:

has the same effect as -s all, i.e. all signals are traced.

-t [!]*syscall*, ...
You specify a comma-separated list of system calls that truss is to trace. To trace all system calls, you specify all for *syscall*. If you put an exclamation mark ! before *syscall*, truss excludes the listed system calls from the trace output. System calls may be specified by name or number [see intro_prm2(2)].

-t [!]*syscall* not specified:

has the same effect as -t all, i.e. all system calls are traced.

-v [!]*syscall*, ...
truss displays the contents of any structures passed by address to the specified system call(s). Input values as well as the values returned by the operating system are shown. For any field used as both input and output, only the output value is given. To trace all system calls, you specify all for *syscall*. If you put an exclamation mark ! before *syscall*, truss excludes the structure contents of the listed system calls from the trace output. System calls may be specified by name or number [see intro_prm2(2)].

-v [!]*syscall* not specified:

has the same effect as -v !all.

-w [!]*fd*, ...
as -r, but for write(2) rather than read(2) .

-x [!]*systemcall*, ...
truss displays the arguments to the specified system call(s) in raw form, usually hexadecimal, rather than symbolically. To specify all system calls, you enter all for *syscall*. If you put an exclamation mark ! before *syscall*, truss excludes the listed system calls from the trace output. System calls may be specified by name or number [see intro_prm2(2)].

-x [!]*systemcall* not specified:

has the same effect as `-x !all`, i.e. no system call is displayed in hexadecimal.

FUNCTIONALITY

If you call `truss` with the `-o` option or redirect standard error to a non-terminal file, `truss` ignores `SIGHUP`, `SIGINT` and `SIGQUIT` signals. This facilitates tracing of interactive programs which receive such signals from the terminal.

If the `truss` output is directed to a terminal or if existing processes are being traced with the `-p` option, `truss` responds to `SIGHUP`, `SIGINT` and `SIGQUIT` signals by releasing all traced processes and terminating. This enables you to terminate excessive trace output and to release processes that existed before `truss` was started. Released processes continue normally as though they had never been touched.

Every machine fault, except a page fault, results in the posting of a signal to the process which caused the fault. A report of a received signal will immediately follow each report of a machine fault, except a page fault, unless that signal is being blocked by the process.

The operating system enforces certain security restrictions on the tracing of processes: Thus, for example, a `truss`, which waits with `semop` for the release of a semaphore, returns a null value instead of the expected "sleeping", as data structures can be lost if a number of processes are traced by `truss`. Thus any command whose object file (`a.out`) cannot be read by a user cannot be traced by that user. Commands for which the set-user-id or set-group-id bit [see `chmod(1)`] has been set can only be traced by the system administrator. Unless run by the system administrator, `truss` loses control of any process which performs an `exec(2)` for an object file for which the set-uid or set-gid bit has been set or which is unreadable. Such processes continue normally, though independently of `truss`, from the point at which the `exec(2)` is performed.

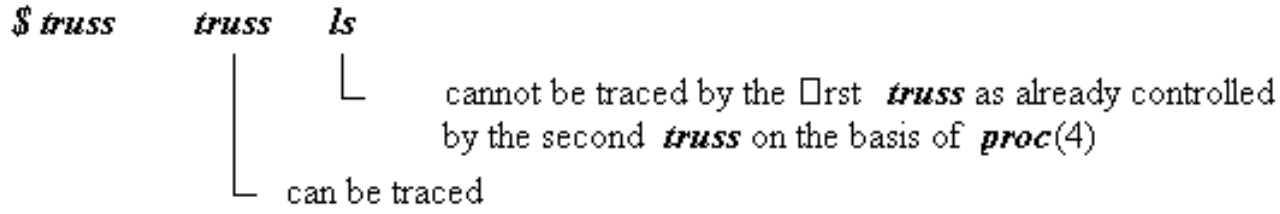
To avoid conflicts with other controlling processes, `truss` will not trace a process which it detects is being controlled by another process via the `/proc` interface. For each existing process, a file that can be accessed by calls like `open(2)`, `close(2)` and `ioctl(2)` is set up in the `/proc` directory. This mechanism is used by debuggers such as `sdb` as an alternative to `ptrace(2)`. `truss` uses this mechanism as well. A process that is already being controlled by another process on the basis of `/proc` cannot be processed by `truss`, as the following example shows:

```
$ truss -o /dev/null sh&
```

Let us assume that the `sh` process has the PID 18027:

```
$ truss -p 18027
truss: someone else is tracing process 18027
```

It is possible, however, to apply `truss` to a debugger that in turn is controlling a process on the basis of `/proc`. `truss` can then trace the debugger, but not the processes started by the debugger. Similarly, `truss` can be applied to another instance of itself, e.g.:



The `truss` output contains tab characters under the assumption that standard tab stops are set every eight positions.

The trace output for multiple processes is not produced in strict time order. For example, a `read(2)` on a pipe may be reported before the corresponding `write(2)`. For any one process, the output is strictly time-ordered.

The system may run out of per-user process slots when tracing of child processes is requested (the system has an upper limit of 25 processes). When tracing more than one process, `truss` runs as one controlling process for each process being traced. For example, the `spell` command requires 9 processes and `truss` adds another 9, giving a total of 18.

`truss` uses `shared memory` and `semaphores` when dealing with more than one process (`-f` and `-p` options with more than one PID). It issues a warning message and proceeds when these are needed but not configured in the system. The `truss` output may become garbled in this case, however, and the output of the `-c` option reports only the top-level command or first PID and no child processes are counted.

ERROR MESSAGES

`truss: someone else is tracing process <PID>`
: `truss` was applied to a process that was already being controlled.

`truss: invalid process id: <PID>`
: Self-explanatory.

`truss: cannot control process <PID>`
: No access rights for process ID *PID*.

`truss: Cannot find program: name`
: Self-explanatory.

EXAMPLES

Example 1

The `find` command is to be traced, with the trace displayed on the screen (standard error) and the output of `find` (standard output) directed to the file `find.out`:

```
$ truss find . -print > find.out
```

Example 2

The system calls `open(2)`, `close(2)`, `read(2)` and `write(2)` invoked by `find` are to be traced:

```
$ truss -t open,close,read,write find . -print > find.out
```

Example 3

The `spell` command is to be traced with a file called `document` passed as an argument. The `truss` output is to be redirected to the file `truss.out`. Since `spell` is a shell script, `truss` must be called with the `-f` option so that any child processes are traced.

```
$ truss -f -o truss.out spell document
```

Example 4

The activities of the `init` process are to be traced. Only the system administrator is allowed to do this.

```
$ truss -p -v all 1
```

FILES

`/proc`
: directory containing the files set up for the current process

SEE ALSO

`ps(1)`, `intro_prm2(2)`, `signal(5)`.

User's Reference Manual

tset(1)

NAME

`tset` - provides information for setting terminal modes

SYNOPSIS

`tset` [*options*] [*type*]

DESCRIPTION

`tset` allows the user to set a terminal's `ERASE` and `KILL` characters, and define the terminal's type and capabilities by creating values for the `TERM` and `TERMCAP` environment variables. If a *type* is given with the `–s` option, `tset` creates information for a terminal of the specified type. The type may be any type given in `/usr/share/lib/termcap`. If the *type* is not specified with the `–s` option, `tset` creates information for a terminal of the type defined by the value of the environment variable, `TERM` unless the `–h` or `–m` option is given. If the `TERM` variable is undefined, `tset` looks in `/usr/share/lib/termcap` for the appropriate information. If these options are used, `tset` searches the `/etc/ttytype` file for the terminal type corresponding to the current serial port; it then creates information for a terminal based on this type. If the serial port is not found in `/etc/ttytype`, the terminal type is set to dumb.

`tset` displays the created information at the standard output. The information is in a form that can be used to set the current environment variables. The exact form depends on the login shell from which `tset` was invoked. The examples below illustrate how to use this information to change the variables.

OPTIONS

`–e`*c* Sets the `erase` character to *c* on all terminals. The default setting is the BACKSPACE, or `CTRL-H`.

`–E`*c* Identical to the `–e` command except that it only operates on terminals that can BACKSPACE.

`–k`[*c*]
Sets the `KILL` character to *c*, defaulting to `CTRL-U`.

`–` Prints the terminal type on the standard output.

`–h` Gives a current idea of terminal type from environment.

`–s` Outputs the `setenv` commands [for `csh(1)`], or `export` and assignment commands [for `sh(1)`]. The type of commands are determined by the user's login shell.

`–S` Only outputs the strings to be placed in the environment variables.

`–r` Sets the `erase` character to DEL, and prints the terminal type.

`–Q` Suppresses the printing of the "`Erase set to`" and "`Kill set to`" messages.

`–I` Suppresses printing of the terminal initialization strings.

`–m`[*port-ID*][*baudrate*]:*type* ...
`tset` is most useful when included in the `.login` [for `csh(1)`] or `.profile` [for `sh(1)`] file executed automatically at login, with `–m` option is given, the first correct mapping prevails.

EXAMPLES

```
tset gt42
tset -mdialup>300:adm3a-mdialup:dw2-Qr-e#
tset -mdial:ti733-mplug:?hp2621-munknown:?-e-k^U
```

To use the information created by the `–s` option for the Bourne shell, (`sh`), repeat these commands:

```
tset -s...>/tmp/tset$$
/tmp/tset$$
rm/tmp/tset$$
```

To use the information create for `csh`, use:

```
set noglobset term=('tset-S...')setenv TERM$term[1]setenv TERMCAP"$term[2]"unset
termunset noglob
```

FILES

`/usr/share/lib/termcap`
terminal capability database

SEE ALSO

`stty`(1), `tty`(1).

User's Reference Manual

tset(1-ucb)
BSD Compatibility Package

NAME

`tset, reset -` establish or restore terminal characteristics

SYNOPSIS

`/usr/ucb/tset [ – ] [ –rsIQS ] [ –e`*C*` ] [ –k`*C*` ] [ –i`*C*` ] [ –m [`*ident* `][ `*test speed* ` ]:`*type*`] ... [ `*type* ` ]`

`/usr/ucb/reset [ – ] [ –rsIQS ] [ –e`*C*` ] [ –k`*C*` ] [ –i`*C*` ] [ –m [`*ident* `][ `*test speed* ` ]:`*type*`] ... [ `*type* ` ]`

DESCRIPTION

`tset` sets up your terminal, typically when you first log in. It does terminal dependent processing such as setting erase and kill characters, setting or resetting delays, sending any sequences needed to properly initialized the terminal, and the like. `tset` first determines the *type* of terminal involved, and then does necessary initializations and mode settings. If a port is not wired permanently to a specific terminal (not hardwired) it is given an appropriate generic identifier such as `dialup`.

`reset` clears the terminal settings by turning off `CBREAK` and `RAW` modes, output delays and parity checking, turns on NEWLINE translation, echo and TAB expansion, and restores undefined special characters to their default state. It then sets the modes as usual, based on the terminal type (which will probably override some of the above). See `stty(1)` for more information. All arguments to `tset` may be used with `reset`. `reset` also uses `rs=` and `rf=` to reset the initialization string and file. This is useful after a program dies and leaves the terminal in a funny state. Often in this situation, characters will not echo as you type them. You may have to type "<LINEFEED>`reset`<LINEFEED>" since <RETURN> may not work.

When no arguments are specified, `tset` reads the terminal type from the `TERM` environment variable and re-initializes the terminal, and performs initialization of mode, environment and other options at login time to determine the terminal type and set up terminal modes.

When used in a startup script (`.profile` for `sh(1)` users or `.login` for `csh(1)` users) it is desirable to give information about the type of terminal you will usually use on ports that are not hardwired. Any of the alternate generic names given in `/etc/termcap` may be used for the identifier. Refer to the `–m` option below for more information. If no mapping applies and a final *type* option, not preceded by a `–m`, is given on the command line then that type is used.

It is usually desirable to return the terminal type, as finally determined by `tset`, and information about the terminal's capabilities, to a shell's environment. This can be done using the `–`, `–s`, or `–S` options.

For the Bourne shell, put this command in your `.profile` file:

`eval `tset –s` *options ...*`

or using the C shell, put this command in your `.login` file:

`eval `tset –s` *options ...*`

With the C shell, it is also convenient to make an alias in your `.cshrc` file:

```
alias tset 'eval `tset –s \!*`'
```

This also allows the command:

```
tset 2621
```

to be invoked at any time to set the terminal and environment. It is not possible to get this aliasing effect with a Bourne shell script, because shell scripts cannot set the environment of their parent. If a process could set its parent's environment, none of this nonsense would be necessary in the first place.

Once the terminal type is known, `tset` sets the terminal driver mode. This normally involves sending an initialization sequence to the terminal, setting the single character erase (and optionally the line-kill (full line erase)) characters, and setting special character delays. TAB and NEWLINE expansion are turned off during transmission of the terminal initialization sequence.

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is `#`, the erase character is changed as if `-e` had been used.

OPTIONS

`-` The name of the terminal finally decided upon is output on the standard output. This is intended to be captured by the shell and placed in the `TERM` environment variable.

`-r` In addition to other actions, reports the terminal type.

`-s` Output commands to set and export `TERM`. `-s` can be used like this:

```
set noglob
eval `tset -s ...`
unset noglob
```

It will bring the terminal information into the environment. Doing so makes programs such as `vi(1)` start up faster. If the `SHELL` environment variable ends with `csh`, C shell commands are output, otherwise Bourne shell commands are output.

`-I` Suppress transmitting terminal-initialization strings.

`-Q` Suppress printing the `Erase set to` and `Kill set to` messages.

`-S` Similar to `-s`, but also provides information for `TERM` and `TERMCAP`. `-S` can be used as follows:

```
set noglob
set term=(`tset -S ...`)
setenv TERM $term[1]
setenv TERMCAP "$term[2]"
unset term
unset noglob
```

`-e`*C* Set the erase character to be the named character *C* on all terminals. Default is the BACKSPACE key on the keyboard, usually "`^H`" (CTRL-H). The character *C* can either be typed directly, or entered using the circumflex-character notation used here.

`-k`*C* Set the line kill character to be the named character *C* on all terminals. Default is "`^U`" (CTRL-U). The kill character is left alone if `-k` is not specified. Control characters can be specified by prefixing the alphabetical character with a circumflex (as in CTRL-U) instead of entering the actual control key itself. This allows you to specify control keys that are currently assigned.

`-i`*C* Set the interrupt character to be the named character *C* on all terminals. Default is "`^C`" (CTRL-C). The character *C* can either be typed directly, or entered using the circumflex-character notation used here.

`-m` [*ident*[*test speed*]`:`*type*] ...

Specify (map) a terminal type when connected to a generic port (such as `dialup` or `plugboard`) identified by the *ident* port ID. The mapping can be performed in accordance with the baudrate *speed*. Any combination of operators is permitted for *test*:

`>` Greater than

`=` or `@` Equals

`<` Less than

`!` It is not the case that (negates the above operators)

`?` Prompt for the terminal type. If no response is given, then *type* is selected by default.

In the following example, the terminal type is set to `adm3a` if the port is a dialup with a speed of greater than 300 or to `dw2` if the port is a dialup at 300 baud or less. In the third case, the question mark preceding the terminal type indicates that the user is to verify the type desired. A `NULL` response indicates that the named type is correct. Otherwise, the user's response is taken to be the type desired.

```
tset -m 'dialup>300:adm3a' -m 'dialup:dw2' -m \
```

```
'plugboard:?adm3a'
```

To prevent interpretation as metacharacters, the entire argument to `-m` should be enclosed in single quotes. When using the C shell, exclamation points should be preceded by a backslash (\).

EXAMPLES

These examples all use the - option. A typical use of `tset` in a `.profile` or `.login` will also use the `-e` and `-k` options, and often the `-Q` option as well. These options have been omitted here to keep the examples short.

To select a 2621, you might put the following sequence of commands in your `.login` file (or `.profile` for Bourne shell users).

```
set noglob
eval `tset -s 2621`
unset noglob
```

If you have a switch which connects to various ports (making it impractical to identify which port you may be connected to), and use various terminals from time to time, you can select from among those terminals according to the *speed* or baud rate. In the example below, `tset` will prompt you for a terminal type if the baud rate is greater than 1200 (say, 9600 for a terminal connected by an RS-232 line), and use a Wyse 50 by default. If the baud rate is less than or equal to 1200, it will select a 2621. Note the placement of the question mark, and the quotes to protect the `>` and `?` from interpretation by the shell.

```
set noglob
eval `tset -s -m 'switch>1200:?wy' -m 'switch<=1200:2621'`
unset noglob
```

The following entry is appropriate if you always dial up, always at the same baud rate, on many different kinds of terminals, and the terminal you use most often is an `adm3a`.

```
set noglob
eval `tset -s \?adm3a`
unset noglob
```

If you want to make the selection based only on the baud rate, you might use the following:

```
set noglob
eval `tset -s -m '>1200:wy' 2621`
unset noglob
```

The following example quietly sets the erase character to BACKSPACE, and kill to CTRL-U. If the port is switched, it selects a Concept 100 for speeds less than or equal to 1200, and asks for the terminal type otherwise (the default in this case is a Wyse 50). If the port is a direct dialup, it selects Concept 100 as the terminal type. If logging in over the ARPANET, the terminal type selected is a Datamedia 2500 terminal or emulator. Note the backslash escaping the NEWLINE at the end of the first line in the example.

```
set noglob
eval `tset -e -k^U -Q -s -m 'switch<=1200:concept100' -m \
     'switch:?wy' -m dialup:concept100 -m arpanet:dm2500`
unset noglob
```

FILES

```
.login
.profile
```

SEE ALSO

csh(1), sh(1), stty(1), vi(1), environ(5).

User's Reference Manual

tty(1)

NAME

`tty -` output path name of current terminal

SYNOPSIS

`tty [-l][-s]`

DESCRIPTION

`tty` outputs the path name of the terminal with which the process is linked. The exit status indicates whether or not standard input is a terminal.

If the process is linked with a virtual terminal, `tty` returns the name of the virtual terminal, not the real terminal.

The specified path name corresponds to that of the `ttyname(3C)` function. (The output format is `"%s\n"`, `<terminalname>`, provided the process is linked with a terminal and the `-s` option was not specified.)

OPTIONS

`-l` If the current terminal is on a synchronous line, the number of the line is output.

`-s` Inhibits output of the terminal path name, returning the exit status only.

Note: You should use `test -t 0` to check whether you can read from the standard input.

`-s` not specified: `tty` reports an error if standard input is not a terminal.

EXIT STATUS

`0` Standard input is a terminal.

`1` Standard input is not a terminal.

`>1` An invalid option was specified or an error occurred.

ERROR MESSAGES

`not on an active synchronous line`

The `-l` option was specified and the standard input is not a terminal on an active synchronous line.

`not a tty`

The standard input is not a terminal and the `-s` option was not specified.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

Output the name of the current terminal:

```
$ tty
/dev/tty03
```

Example 2

The following script is to send output to the screen even if standard output is redirected to a file:

.

```
  .
echo 'Output to the terminal' > `tty`
  .
  .
```

Example 3

The following script is to generate an error message if standard input is not the terminal:

```
  .
  .
if tty -s
then
read input
  .
  .
else
echo 'Standard input is not a terminal' >&2
fi
  .
  .
```

SEE ALSO

ttyname(3C) .

User's Reference Manual

type(1)

NAME

`type` - query command type

SYNOPSIS

`type` *name* ...

DESCRIPTION

The Bourne shell built-in `type` is used to indicate how the shell would interpret a given name if it were entered as a command. The command type (and some other information) is written to standard output.

In the Korn shell `ksh(1)`, `type` is an exported alias for `whence -v`; `whence(1)` is a Korn shell built-in command.

OPERANDS

name Name of the command whose type is to be queried. Several blank-separated names may be specified at a time.

If *name* is neither a shell function nor a shell built-in, the shell searches for an executable file with this name in the directories whose path names are defined in the `PATH` variable.

The `type` command indicates whether *name* is a shell function or a shell built-in. If *name* designates an executable file, the absolute path name or a path name in the form `./`*name* is displayed.

If *name* is a shell function, `type` will additionally print the function definition. In the Korn shell, the definition is not shown.

If *name* is not an executable file, `type` issues an error message. The same error message is also output if *name* does not exist in the directories whose path names are assigned to the `PATH` variable.

ERROR MESSAGES

`name not found`

This error message means:

- either there is no file with this name in the directories that can be accessed via the `PATH` variable,
- or the file with this name is not executable.

ENVIRONMENT VARIABLES

`PATH` Search path of the shell.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The following series of commands illustrates the kind of information supplied by the shell built-in `type`:

```
$ type tar
tar is /bin/tar
$ type type
type is a shell builtin
$ type ll
ll is a function
```

```
ll(){
ls –al $*  pg
}
$ type typetest
typetest is ./typetest
```

NOTES

`type` exists both as an external command (`/usr/bin/type`) and as a built-in shell command in the Bourne shell `sh`(1) . The shell generates a new process to execute `/usr/bin/type`.

Some differences in behavior may occur when using `type`, depending on which command is being used. The possible differences are not described specifically.

SEE ALSO

`command`(1) , `sh`(1) , `whence`(1) .

User's Reference Manual

ul(1)
BSD Compatibility Package

NAME

`ul` - underline

SYNOPSIS

`/usr/ucb/ul` [`–i`] [`–t` *terminal*] [*filename* ...]

DESCRIPTION

The `ul` command reads the named *filename*s (or standard input if none is given) and translates occurrences of underscores to the sequence which indicates underlining for the terminal in use, as specified by the environment variable `TERM`. `ul` uses the `/usr/share/lib/termcap` file to determine the appropriate sequences for underlining. If the terminal is incapable of underlining, but is capable of a standout mode then that is used instead. If the terminal can overstrike, or handles underlining automatically, `ul` degenerates to `cat`. If the terminal cannot underline, underlining is ignored.

OPTIONS

`–t` *terminal*
: Override the terminal kind specified in the environment. If the terminal cannot underline, underlining is ignored.

`–i`
: Indicate underlining by a separate line containing appropriate dashes `–`; this is useful when you want to look at the underlining which is present in an `nroff` output stream on a CRT-terminal.

SEE ALSO

`cat`(1), `man`(1), `nroff`(1M).

User's Reference Manual

ulimit(1)

NAME

`ulimit -` set or display user file size limits

SYNOPSIS

Format 1

`ulimit [ –H ] [ –S ] [ option ... ]`

Format 2

`ulimit [ –H ] [ –S ] [ option ... ] limit`

DESCRIPTION

The shell built-in `ulimit` enables you to

- check the file size limits imposed for the current shell and its child processes.
- set or change the individual file size limits for the current shell and all its child processes. Normal users without system administrator privileges may decrease this limit, but not increase it. The new limits apply to files written by the current shell and all its child processes.

  You cannot increase a limit that has been decreased until you have terminated the shell in which you have decreased the limit.

  The following limits, described in more detail under `getrlimit(2)`, are available for your current shell and all its child processes:

  `c – core size`
  : Maximum core dump size in the `core` file (in 512-byte blocks), if a process was aborted with errors [see `signal(5)`].

    If core size is 0, no `core` file is created.

  `d – data segment`
  : Maximum size of a data segment or "heap" (in Kbytes) in a process.

  `f – file size`
  : Maximum file size (in 512-byte blocks) that you may create (write); there is no limit on reading. If file size is 0, no files can be created. If you exceed the default value, you either receive an error message from the appropriate command (depending on the command you have used to create the file) or the new file only contains as much data as the imposed limit can accommodate.

    Example for file size:

    After the command `ls –lR >file`, `file` will contain as many bytes as are permitted by the current size limit.

    The command `cp(1)`, on the other hand, will issue the error message `bad copy to file` if the file to be copied exceeds the currently set file size limit.

  `n – number of file descriptors`
  : Maximum number of (open) file descriptors permitted in a process plus 1.

  `s – stack size`
  : Maximum stack segment size (in Kbytes) permitted in a process.

  `t – time`
  : Maximum CPU time (in seconds) permitted for a process.

  `v – virtual memory size`
  : Maximum virtual memory size (in Kbytes) permitted in a process.

OPTIONS

Format 1: Check limits

`ulimit` [–H] [–S] [*option* ...]

`ulimit` writes the limits checked by *option* to standard output.

–H Checks a `hard` limit.

–S Checks a `soft` limit.

Neither –H nor –S specified:

`ulimit` writes the `soft` limits to standard output. You can use options to specify the limits to be checked. You can combine the options however you want.

No option specified

`ulimit` uses the –f option (see **Format 2**).

–a Checks all limits.

The other options are described under **Format 2**.

Format 2: Set limits

`ulimit` [–H] [–S] [*option* ...] *limit*

`ulimit` sets the limit defined by *option* to *limit*. You can only set one new limit per command call.

–H Sets a `hard` limit. Normal users without administrator privileges may reduce any `hard` limit. However, only the system administrator may increase a `hard` limit.

–S Sets a `soft` limit. Any user can set a `soft` limit to a value less than the `hard` limit.

Neither –H nor –S specified:

`ulimit` sets `hard` and `soft` limits to the specified value. You can use options to specify the limits to be set.

No option specified

`ulimit` uses the –f option.

–c Maximum core dump size in the `core` file (in 512-byte blocks).

–d Maximum size of the data segment or `heap` (in Kbytes).

–f Maximum file size (in 512-byte blocks).

–n Maximum number of file descriptors plus 1.

–s Maximum stack segment size (in Kbytes).

–t Maximum CPU time (in seconds).

–v Maximum virtual memory size (in Kbytes).

limit

Sets the size limit for the current shell and all its child processes in accordance with the specified option. Normal users without system administrator privileges may only specify values less than the current size limit defined for *limit*. As a system administrator, however, you may also increase this limit with *limit*. If you specify the string `unlimited` for *limit*, the limit is set to the maximum possible value.

ERROR MESSAGES

`Bad ulimit`

You have tried to increase the currently set file size limit. This privilege is reserved for system administrators only.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

When `ulimit` is invoked as in **Format 1**, all the current limits are displayed. The soft limit is always located in front of the colon, the hard limit after the colon.

```
$ ulimit –Sa
time(seconds) unlimited
file(blocks) unlimited
data(kbytes) 16384
stack(kbytes) 16384
coredump(blocks) 2048
nofiles(descriptors) 64
memory(kbytes) 16384
```

Example 2

You want to increase the current limit for the maximum number of file descriptors. Any user can do this provided the new limit is lower than the hard limit. The new limit is also valid in subshells. (The output is only valid for the Bourne shell `sh`.)

```
$ ulimit –Sn 80
$ ulimit –HSn
80:1024
```

Example 3

You want to check and then reduce the maximum file size. This limit can only be increased again by the system administrator.

```
$ ulimit
4194303
$ ulimit 20000
$ sh
$ ulimit
20000
```

The new limit is also valid in subshells. From now on, only files smaller than 20,000 * 512 bytes can be created.

NOTES

Some differences in behavior may occur when using `ulimit`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh(1)`, `sh(1)`, `getrlimit(2)`, `ulimit(2)`, `signal(5)`.

User's Reference Manual

umask(1)

NAME

`umask` - set file-creation mode mask

SYNOPSIS

`umask` [*mask*]

DESCRIPTION

The shell built-in `umask` sets or displays the current user file-creation mode mask. This mask defines the access permissions to be assigned to all new files and directories which you subsequently create in the current shell or in one of its subshells.

Changes made in the file-creation mode mask remain in effect until a new value is set with `umask` or the shell in which `umask` was originally called is terminated.

The system administrator can use `umask` to define the value of the file-creation mode mask in the `/etc/profile` file. Since `/etc/profile` is executed by every login shell, the access permissions set in this way are valid for every used logged in on the system (see *Example 1*).

Setting permissions with the file-creation mode mask

When creating files and directories the following permissions are generally assigned as base settings [see `open(2)`]:

- `rw-rw-rw-` to files, i.e. 110110110 in binary notation
- `rwxrwxrwx` to directories, i.e. 111111111 in binary notation.

`umask` is only capable of withholding existing permissions from these base settings. This means that you cannot use `umask` to have execute permission automatically assigned to new files. You may, however, set the appropriate x-bit with the `chmod(1)` command.

The file-creation mode mask comprises three octal digits, which are specified when calling `umask`. The permissions they refer to are derived as follows:

1. Convert the digits in the mask to their binary equivalents.
2. Create the complement to each of these binary numbers, i.e. replace the zeros by ones, and ones by zeros.
3. Add this binary complement to the binary value of the base mode setting by logically ANDing the two; the result thus only contains ones in positions where both pairs have ones; in all other positions it contains zeros.

Example:

The file-creation mode mask 022 changes permissions as follows:

– Files:

The base mode setting for files is 110110110.

1. The binary equivalent of octal 022 is	000010010
2. The complement of this value is	111101101
3. AND operation:	111101101
	110110110
	110100100

The permissions for all newly created files will thus be `rw-r--r--`

– Directories:

The base mode setting for directories is 111111111.

AND operation: 111101101
111111111
111101101

All new directories created will have the permissions `rwxr-xr-x`

OPERANDS

mask Three octal digits comprising the file-creation mode mask. This mask defines the permissions to be assigned to all new files or directories that the user subsequently creates in the current shell or in any of its subshells (see **Setting permissions with the file-creation mode mask** above).

Since `umask` can only withhold existing permissions from the base settings, you cannot use it to have execute permissions directly assigned to files, regardless of what you specify for `mask`. Use the command `chmod(1)` instead.

In the Korn shell `ksh` you can specify the file-creation mode mask symbolically [see `ksh(1)`].

mask not specified:

`umask` displays the current user file-creation mode mask. The output can be used as input for further `umask` commands and has the following format: `0`*nnn*

`0` Indicates octal notation.

nnn The current file-creation mode mask in octal.

In the Korn shell `ksh` the file-creation mode mask can also be output symbolically [see `ksh(1)`].

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The following lines are often included in the `/etc/profile` file:

```
if [ "$USER" != "admin" -a "$USER" != "root" ]
then
        umask 066
fi
```

Since every login shell executes the `/etc/profile` file, the file-creation mode mask 066 applies to all users except `root` and `admin`.

The following permissions are thus assigned by default:

- for new files created: `rw-------`
- for new directories created: `rwx--x--x`

Example 2

Change the file-creation mode mask and display it:

```
$ umask 033
$ > new
$ mkdir newdir
$ ls -ld new newdir
-rw-r--r--   1 anne     other          0   Mar 22 09:49 new
```

```
drwxr--r--   2 anne     other        520   Mar 22 16:40 newdir
$ umask
0033
```

The output of the `ls –ld ...` command shows which access permissions are assigned to new files and directories when the file-creation mode mask has been set to the value 033.

NOTES

Some differences in behavior may occur when using `umask`, depending on which shell is being used. The possible differences are not described specifically.

FILES

`/etc/profile`

File executed by each login shell; used to set a shell environment.

The system administrator normally defines one mask value for users without special privileges and one for `root` in this file.

SEE ALSO

`chmod`(1), `ksh`(1), `sh`(1), `chmod`(2), `creat`(2), `open`(2), `umask`(2), `profile`(4).

uname(1)

NAME

`uname` - output or set name of current system

SYNOPSIS

Format 1

`uname` [*option ...*]

Format 2

`uname –S` *nodename*

DESCRIPTION

The `uname` command writes information related to the current operating system to standard output.

OPTIONS

Format 1: Output the system name

`uname` [*option ...*]

No option specified:

The name of the operating system is output, e.g. `SINIX-Y`.

`–a` (a – all) Outputs the following information: the name of the operating system, the system's node name, the version number of the operating system, the revision status of the operating system version, the machine hardware name, the number of CPUs, the megabytes of main memory, and the processor type, e.g. `SINIX-Y utmost 5.43 B0032 RM600 6/256 R4000`.

`–M` Outputs the number of CPUs and megabytes of memory, e.g. `6/256`.

`–m` (m – machine type) Outputs the machine hardware name, e.g. `RM600`.

`–n` (n – node) Outputs the node name of the operating system. This is the name by which a system is known in a communication network, e.g. `utmost`.

`–p` (p – processor type) Outputs the processor family of the system that you are currently using, e.g. `R4000` for RISC-based systems.

`–r` (r – release) Outputs the version number of the operating system, e.g. `5.43`.

`–s` (s – system) Outputs the name of the operating system. This is the name by which the operating system is known in the local installation, e.g. `SINIX-Y`.

`–v` (v – version) Outputs the revision status of the operating system version, e.g. `B0032`.

Format 2: Set and change the node name

`uname –S` *nodename*

Format 2 may be used by the system administrator only.

`–S` *nodename*

The node name of the system is changed to *nodename*.

The command `uname –S` outputs the following message:

```
uname can't change the node name in the whole software.
For more details refer to the uname manual page.
```

This means:

If the node name of the system is also the name of one or more interfaces, you have to update the network configuration and restart all network services.

Note:

The node name of a system can also be changed using sysadm(1M) , menu item "system_setup".

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

NOTES

A user wishing to maintain backwards compatibility with older releases must set the environment variables correctly to ensure that uname supplies the old name. There are various possibilities for doing this.

1) The /etc/default/uname file exists.

In this case, the $_SNI_NAME environment variable is evaluated in accordance with the /etc/default/uname file and can be set locally by any user.

a) If this variable is set to _SNI_NAME=comp (default), or to a value other than "std", uname –s returns "SINIX-Y"or "SINIX-N".

b) If this variable is set to _SNI_NAME=std, uname –s returns "ReliantUNIX-Y" or "ReliantUNIX-N".

2) The /etc/default/uname file does not exist.

In this case, the four environment variables OLD_SYSNAME, OLD_RELEASE, OLD_MACHINE, OLD_VERSION can be set, in order to change the default values displayed for the system, release, machine, and version name (e.g. using /etc/profile). These variables can also be set locally by any user (e.g. by means of the .profile file).

Default output corresponds to the output with _SNI_NAME=comp.

EXAMPLES

Format 1: Example of using compatibility variables

1. You want to ensure that uname always returns the old name: Set _SNI_NAME=comp in the /etc/default/uname file.
2. You want to ensure that uname always returns the new name: Set _SNI_NAME=std in the /etc/default/uname file.
3. You want to ensure that in certain cases (users, applications) uname returns output that differs from the setting in /etc/default/uname: For example, for the old value set

   ```
   _SNI_NAME=comp
   export _SNI_NAME
   ```

   in the application or user environment.
4. You want to replace all uname output with your own values: Delete the /etc/default/uname file and use the OLD_... environment variables for controlling your uname output.

Format 2: Example of querying the node name

You want to find out the node name of the operating system as well as the version number of the operating system you are running:

```
$ uname –nr
utmost 5.43
```

FILES

/etc/default/uname

SEE ALSO

sysinfo(2), uname(2).

User's Reference Manual

uncompress(1)

NAME

uncompress - expand compressed files

SYNOPSIS

uncompress [*option ...*] [*file ...*]

DESCRIPTION

uncompress expands files compressed by compress. The files are restored to their original form. The access permissions and access and modification dates for the specified files remain unchanged. On systems which evaluate the RSTCHOWN variable, the user who invokes uncompress is made the owner of the expanded file (unless the invoking user is the system administrator). On other systems the owner stays the same.

Whenever possible, each file specified is replaced by a file with the same name but without the .Z extension.

Expansion is not performed if

- the file to be created already exists and uncompress is running in the background,
- the specified file was compressed with a higher value for maxbits (see compress –b) than your current system can process,
- there are links to the compressed file.

OPTIONS

No option specified:
The specified files are expanded.

–c uncompress writes the expanded files to standard output. No files are modified or created. uncompress –c has exactly the same effect as zcat(1).

–f (force) *file*.Z is expanded even if a file named *file* already exists. *file* is overwritten.

–f not specified:

uncompress asks whether the existing file should be overwritten or not. You are not asked for confirmation, however, when uncompress is running in the background. The specified file is not expanded.

–v (verbose) uncompress issues a message confirming successful expansion: *file*.Z: –– replaced with *file*.

–– If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file Name of a file compressed with compress. You can specify more than one file. If you specify a dash – for one of the files, uncompress reads from the standard input. Each file name must include a .Z suffix which you do not have to specify when invoking uncompress. The expanded file is assigned the name *file*. *file*.Z is deleted after being successfully expanded (except in the case of –c). Access permissions, access and modification dates, and the owner of the file remain unchanged.

On systems which evaluate the RSTCHOWN variable, the user who invokes uncompress is made the owner of the expanded file (unless the invoking user is the system administrator). On other systems the owner stays the same.

EXIT STATUS

0 successful

>0 not successful

ERROR MESSAGES

file.Z: not in compressed format
: The specified file is not available in compressed format or is a directory.

file.Z: compressed with *xx* bits, can only handle *yy* bits
: The specified file was compressed with too high a value for maxbits (see compress –b). The system on which you called uncompress cannot process this value. Compress the file again using the value specified as *yy* in the error message or with a lower value.

file.Z: –– has *xx* links: unchanged
: There are *xx* links to the specified file. The file is not expanded.

uncompress: corrupt input
: The SIGSEGV signal (segmentation violation; an addressing error caused by unauthorized segment access) was received, which normally indicates that the input file is corrupted.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

Answers to yes/no queries must be given in the language appropriate to the current locale.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

You want to expand the file topsecret.Z. The same directory already contains a file called topsecret, but this can be overwritten.

```
$ ls –l
total 62
-rw-------  1  felix    group1    4862 Mar 12 10:16 topsecret
-rw-------  1  felix    group1   26326 Feb 08 10:27 topsecret.Z
$ uncompress –v topsecret.Z
topsecret already exists; do you wish to overwrite topsecret (y or n)? y
topsecret.Z: –– replaced with topsecret
$ ls –l
total 138
-rw-------  1  felix    group1   69845 Feb 08 10:27 topsecret
```

SEE ALSO

compress(1), pack(1), pcat(1), unpack(1), zcat(1).

User's Reference Manual

unexpand(1)

NAME

`unexpand` - convert spaces to tabs

SYNOPSIS

`/usr/ucb/unexpand` [`–a` | `–t` *tablist*] [*file* ...]

DESCRIPTION

The `unexpand` commands writes files or standard input to standard output, with blanks at the beginning of each line converted to the maximum number of tabs, followed by the minimum number of blanks needed to pad the line to the next tab stop, which were originally filled by the converted characters. A tab stop is set after every 8 column positions by default. All backspace characters are copied to the output and cause the column position count for for tab stop calculations to be decremented. The column position count will not be decremented below zero.

OPTIONS

`–a` Not only are the blanks at the start of each line converted; rather all sequences of two or more blanks directly in front of a tab stop are converted into the maximum number of tabs followed by the minimum number of blanks needed to pad the line to the next tab stop, which were originally filled by the converted characters.

`–t` *tablist*
Specifies the tab stops. The argument *tablist* must consist of one or more numbers, separated by blanks or commas, in ascending order. A list separated by blanks must be enclosed in quotes. If only one number is specified, the tab stops will be set to *tablist* column positions instead of the default 8 column positions. If multiple numbers are given, the tabs will be set at the specified column positions.

Each tab stop position *N* must be an integer value greater than zero, and the specifications must be in ascending order. This means that tabbing from the start of the line of output to position *N* causes the next character output to be in the (*N*+1)th column position in that line.

For characters that are one position behind the last position defined in a list with multiple tab stops, there is no conversion of blanks into tabs.

If `–t` is specified, conversion of blanks is not restricted to the leading blanks. `–a` is ignored in this case.

file The file whose blanks are to be replaced by tab characters.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of `unexpand`:

`LANG` Specifies a default value for the internationalization variables that are unset or null. If `LANG` is unset or null, the corresponding default value from the internationalized environment is used. If one of the internationalization variables contains an invalid setting, the command behaves as if none of the variables has been defined.

`LC_ALL`
If this variable has been assigned a value, i.e. it is not a null string, this value overrides the values of all the other internationalization variables.

`LC_CTYPE`
Determines the internationalized environment for the interpretation of byte sequences (e.g. single-byte characters as opposed to multi-byte characters in arguments and input files). The internationalized environment for the processing of tab characters and blanks and for the specification of the width in column positions each character would occupy on a constant-width output device.

`LC_MESSAGES`
Determines the format and contents of error messages.

NLSPATH
: Determines the position of message catalogs for the processing of LC_MESSAGES.

STANDARD OUTPUT (STDOUT)

Standard output is equivalent to the input files with blanks converted into the appropriate number of tabs.

EXIT STATUS

0 Successful completion.

>0 An error occurred.

APPLICATION USAGE

The default behavior of unexpand, whereby only leading blanks are taken into consideration, may not be desired in some cases. Users who always want to convert all blanks in a file should use an alias definition [see ksh(1)] to create a version of unexpand that is always called with the options –a or –t 8.

SEE ALSO

expand(1) , ksh(1) , tabs(1) .

User's Reference Manual

uniq(1)

NAME

uniq - report repeated lines

SYNOPSIS

uniq [*option ...*] [*input_file* [*output_file*]]

DESCRIPTION

The uniq command searches a file for sequences of identical lines and writes the file to standard output, removing all but one of repeated lines in the process. Note that repeated lines must be adjacent in order to be found, i.e. the input file must be sorted.

OPTIONS

The options −c, −d, and −u must not be combined.

No option specified:
The named *input_file* is output without repeated lines.

−c Outputs all lines without repetitions, starting each line with a decimal number to indicate how often it occurred repeatedly in *input_file*. Counts are printed right-justified up to column 4; actual lines begin on column 6. uniq ignores the −u and −d options if set with the −c option.

−d Outputs one copy each of only those lines that are repeated in *input_file*.

−s *n* Causes the first *n* characters from the beginning of the line to be ignored when comparing for duplicates.

If the −s option is combined with the −f option, the first *n* characters after the *m*th field are ignored. Blanks following the *m*th field are not ignored: they must be allowed for in the value of *n*.

This corresponds to the old option +*n*, which is still supported, but must not be combined with the new synopsis (−f or −s).

−s not specified:

Lines are compared from the beginning of the line or beginning with field *m*+1 (option −f).

−f *m* Ignores the first *m* fields from the beginning of the line, plus any tabs or blanks located in front of a field, when comparing for duplicates. A field is a string of non-blank characters separated from its neighbors by tabs or blanks.

This corresponds to the old option −*m*, which is still supported, but must not be combined with the new synopsis (−f or −s).

−f not specified:

Lines are compared from the beginning of the line or beginning with character *n*+1 (option −s).

−u Outputs only the lines that are not repeated in *input_file*.

−− If *input_file* begins with a dash (−), the end of the command line options must be marked with −−.

input_file
Name of the file that is to be examined. If you specify a dash − for *input_file*, uniq reads from the standard input.

input_file not specified: uniq reads from standard input.

output_file
Name of the file to which the output is to be written. If you specify a dash − for *output_file*, uniq writes to the standard output.

output_file not specified: uniq writes to standard output.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

LC_CTYPE governs character classes and character conversion (shifting).

If LC_MESSAGES or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

You want to search a file for identical lines, regardless of where they are located in the file. A count showing how often each of these lines occurs is also to be output.

```
$ sort file | uniq -c
```

Example 2

You want to output the 10 most frequently occurring words in the file text.

```
$ cat text | sed 's/[      ][      ]*/\
> /g' | sed '/^$/d' | sort | uniq -c | sort -rn | head
```

Explanation:

- The first sed call generates a list of all words from text, by replacing consecutive tabs or blanks by a newline character. The square brackets each contain a tab and a blank.
- The second sed call deletes all blank lines.
- sort sorts the generated list in ASCII collating sequence.
- uniq -c removes duplicate lines from the sorted list and precedes each remaining line with a count indicating its frequency of occurrence.
- sort -rn does a reverse sort on this frequency list, i.e. the most frequent line appears first; the line with the least number of repetitions appears last.
- head prints the first 10 lines of the list.

SEE ALSO

comm(1), sort(1).

User's Reference Manual

units(1)

NAME

`units` - conversion program

SYNOPSIS

`units`

DESCRIPTION

`units` computes conversion factors for units given in different scales.

FUNCTIONALITY

`units` outputs the conversion factors required for two units at a time. It works interactively, i.e. prompts you to enter the first unit as soon as you issue the command:

```
$ units
you have:
```

On entering the first unit, you will be prompted again for the second unit:

```
you want:
```

After you specify the second unit, `units` will print the required conversion factors. The first factor indicates the number with which the first unit must be multiplied in order to obtain the second unit; this factor is preceded by an asterisk (*).

The second conversion factor indicates the number with which the second unit must be multiplied in order to obtain the first unit. A slash (/) precedes this factor. Both conversion factors are printed in scientific notation with six decimal places. For example: 9.797299e-01 represents $9.797299 * 10^{-1}$, i.e. 0.9797299.

Example:

```
$ units
you have: inch
you want: cm
        * 2.540000e+00
        / 3.937008e-01
```

The units to be converted can be defined by arbitrarily combining other units as follows:

- Units separated by one or more blanks are multiplied.
- Units separated by a slash / are divided.
- Powers are indicated by suffixed positive integers.
- Preceding positive integers are interpreted as numeric multipliers.

Example

```
$ you have: 15 lbs force/in2
  you want: atm
          * 1.020689e+00
          / 9.797299e-01
```

`units` can only compute conversions that can be expressed in terms of a product; thus, it cannot compute a conversion factor from Celsius to Fahrenheit, for example.

As well as common units and their abbreviations, `units` recognizes a number of constants from the fields of chemistry, physics and mathematics:

`pi` ratio of circumference to diameter

`c` speed of light

`e` charge of an electron

`g` acceleration of gravity

`force`
: same as `g`

`mole` Avogadro's number

The `/usr/share/lib/unittab` file includes a comprehensive table of all units and constants known to `units` as well as the conversion factors used (but note that the foreign currency exchange rates are out of date).

For the system administrator only:
: The system administrator may edit the file `/usr/share/lib/unittab` in order, for example, to update the conversion factors for the various currencies or to add new units and their conversion factors.

Striking the <DEL> or <CTRL-D> key terminates the execution of `units`.

ERROR MESSAGES

`cannot recognize` *xyz*

You have specified a unit that is unknown to the `units` command. Check the valid `units` in the `/usr/share/lib/unittab` file.

`conformability`

`units` cannot calculate a conversion factor for the given units. The definitions of the specified units are also output with this error message.

EXAMPLES

Example 1

Convert miles to kilometers:

```
$ units
you have: mile
you want: km
        * 1.609344e+00
        / 6.213712e-01<CTRL-D>$
```

Example 2

Convert bars to atmospheres:

```
$ units
you have: bar
you want: atm
        * 9.869233e-01
        / 1.013250e+00<CTRL-D>$
```

Example 3

Convert gallons to barrels:

```
$ units
you have: gallon
you want: barrel
        * 2.380952e-02
        / 4.200000e+01<CTRL-D>$
```

FILES

`/usr/share/lib/unittab`
: Contains a complete list of all units and constants recognized by the `units` command.

User's Reference Manual

unpack(1)

NAME

unpack - expand compressed files

SYNOPSIS

unpack *file* ...

DESCRIPTION

unpack expands files compressed by pack(1) to their original form.

Note: You can also compress files with compress(1) and uncompress them again using uncompress(1).

OPERANDS

file Name of a file that was compressed with pack. You may specify the name of the file with or without the .z extension; however, the name must end in .z.

If the specified file is a packed file, it will be replaced by its expanded version. The new file has the .z extension stripped from its name and receives the same access permissions and access and modification dates as the packed file.

On systems in which the RSTCHOWN variable is set, the user who invokes unpack is made the owner of the expanded file (unless the invoking user is the system administrator). On other systems the owner stays the same.

If there are links to the compressed file, unpack gives the warning:

unpack: *file*.z: Warning: file has links

but the file is nevertheless expanded.

EXIT STATUS

0 Successful termination.

>0 An error occurred. In this case, *file* is not created and the original file is retained.

ERROR MESSAGES

All the errors described below result in the failure of the unpack command, i.e. the file remains in compressed form.

unpack: *file*: already exists
: There is already a file with the name that the packed file would be assigned after expansion.

unpack: *file*: cannot create
: The unpacked file cannot be created because you have no write permission for the directory in which this file is to be created.

unpack: *file*.z: cannot open
: You have no read permission for the packed file *file*.z, or *file*.z does not exist.

unpack: *very_long_filename*: file name too long
: The file name contains more than {NAME_MAX} bytes.

unpack: *file*.z: not in packed format
: *file*.z is not a file compressed with pack.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other

environment variables that affect internationalization.

EXAMPLES

The file `catherine` is first compressed with `pack` and then expanded with `unpack`:

```
$ ls –l
total 532
-rw-------   1 anne     other     268716   Jan 05 10:03 catherine
$ pack catherine
pack: catherine: 41.7% Compression
$ ls –l
total 312
-rw-------   1 anne     other     156692   Jan 05 10:03 catherine.z
$ unpack catherine
unpack: catherine: unpacked
$ ls –l
total 532
-rw-------   1 anne     other     268716   Jan 05 10:03 catherine
```

SEE ALSO

`cat(1)`, `compress(1)`, `pack(1)`, `pcat(1)`, `uncompress(1)`, `zcat(1)`.

User's Reference Manual

unset(1)

NAME

`unset` - delete shell variables or shell functions from theenvironment

SYNOPSIS

`unset` *name* ...

DESCRIPTION

The shell built-in `unset` removes the specified shell function or shell variable from the current environment.

The built-in environment variables `IFS`, `MAILCHECK`, `PATH`, `PS1`, and `PS2` cannot be deleted using `unset`.

OPERANDS

name Name of the shell variable or function to be removed from the current environment. Several blank-separated names may be specified in a single command line.

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire locale. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLE

Remove the shell function `ll` (for the Korn shell `ksh` you must specify `unset –f ll`):

```
$ type ll
ll is a function
ll(){
ls –al $* | pg
}
$ unset ll
$ ll
ll not found
```

NOTES

Some differences in behavior may occur when using `unset`, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

`ksh(1)`, `set(1)`, `sh(1)`.

User's Reference Manual

uptime(1)
BSD Compatibility Package

NAME

`uptime -` show how long the system has been up

SYNOPSIS

`/usr/ucb/uptime`

DESCRIPTION

The `uptime` command shows the current time of day, how long the system has been up, the number of users logged into the system and the load averages. The load average numbers give the number of jobs in the run queue averaged over 1, 5, and 15 minutes [cf. `sar(1)` , `–q` option and `mpstat(1M)`].

The output corresponds to the first line of the `w(1)` command's output.

EXAMPLES

Below is an example of the output `uptime` provides:

```
uptime
7:53am up 20:14, 9 users, load average: 2.25, 66.57, 163.74
```

NOTES

`who –b` gives the time the system was last booted.

SEE ALSO

`ps(1)`, `sar(1)`, `w(1)`, `who(1)`, `mpstat(1M)`, `whodo(1M)`.

User's Reference Manual

users(1)
BSD Compatibility Package

NAME

`users` - display a compact list of users who are currently logged in

SYNOPSIS

`/usr/ucb/users` [*file*]

DESCRIPTION

The `users` command lists the login names of the users currently logged in to the system in alphabetical order.

Specifying *file* tells `users` where to find its information; by default it checks `/var/adm/utmp`.

Typing `users` is similar to typing `who –q`. In the case of `who –q`, however, the users currently logged in are listed in chronological order and a total number is output.

EXAMPLE

```
$ users
george john paul ringo
```

FILES

`/var/adm/utmp`

SEE ALSO

`who`(1) .

User's Reference Manual

vacation(1)

NAME

`vacation -` automatically respond to incoming mail messages

SYNOPSIS

`vacation` [`–l` *logfile*] [`–m` *mailfile*] [`–M` *replyfile*] [`–F` *fallback*] [`–d`]

DESCRIPTION

When new mail arrives in a mailbox, the `mail` command first checks if there is any indication that mail is to be forwarded to another recipient or piped as input to another command [see `mail(1)`]. The purpose of `vacation` is to leave a note in a mailbox to indicate that mail is to be forwarded to an alternate (vacation) mailbox and a standard response is to be sent to the mail's originator. The mail message is then saved in the alternate mailbox.

OPTIONS

`–l` *logfile*

File to which `vacation` writes the names of mail originators who have already been sent the standard response.

`–l` *logfile* not specified:

`vacation` uses the file `$HOME/.maillog`.

`–m` *mailfile*

Alternate mailbox in which mail is to be saved. You should not use `/var/mail/`*login_name*, the name of the default mail file, as this file can only hold either mail messages or indications of mail forwarding but not both simultaneously.

`–m` *mailfile* not specified:

`vacation` uses the file `$HOME/.mailfile`.

`–M` *replyfile*

File containing the standard reply automatically sent to the mail originator.

`–M` *replyfile* not specified:

`vacation` uses the file `/usr/share/lib/mail/std_vac_msg`, which contains:

```
Subject: AUTOANSWERED!!!

I am on vacation. I will read (and answer if necessary)
your e-mail message when I return.
This message was generated automatically and you will
receive it only once, although all the messages you send
me while I am away WILL be saved.
```

`–F` *fallback*

As *fallback* you can name a user (login name) for mail to be forwarded to if there are problems in delivering it to your vacation mailfile. This prevents mail being returned to the sender.

`–F` *fallback* not specified:

If there are problems, mail is returned to sender.

`–d` *logfile* will have the day's date appended.

To cancel `vacation` when you return to the office, you enter:

```
$ mail -F ""
```

EXAMPLES

While you are away, you want to have your mail saved in `/usr/mail/holiday`. You want anyone who

writes to you to be sent the reply which you have stored in `/usr/mail/autoreply`. The login name of each mail originator who has seen this reply is to be listed in `/usr/mail/sender/autoreply`.

```
$ vacation –l /usr/mail/originator/reply –m /usr/mail/holiday \
  –M /usr/mail/reply
```

FILES

`$HOME/.maillog`
: By default holds the names of mail originators who have already been sent the standard response

`$HOME/.mailfile`
: Default vacation mailbox

`/var/mail/`*`login_name`*
: Users' standard mailboxes

`/usr/share/lib/mail/std_vac_msg`
: Contains the standard reply

`/tmp/notif*`
: Temporary storage file

`/usr/lib/mail/vacation2`
: Shell script invoked by the `vacation` command

SEE ALSO

`mail(1)`.

User's Reference Manual

vacation(1-ucb)

NAME

`vacation -` reply to mail automatically

SYNOPSIS

`/usr/ucb/vacation [ –I ]`

`/usr/ucb/vacation [ –j ] [ –a`*alias* `] [ –t`*N* `] `*username*

DESCRIPTION

`vacation` automatically replies to incoming mail. The reply is contained in the file `.vacation.msg`, that you create in your home directory.

This file should include a header with at least a "`Subject:`" line (it should not include a "`From:`" or a "`To:`" line). For example:

```
Subject: I am on vacation
I am on vacation until July 22. If you have something urgent,
please contact Joe Jones (jones@f40).
 --John
```

If the string `$SUBJECT` appears in the `.vacation.msg` file, it is replaced with the subject of the original message when the reply is sent. A `.vacation.msg` file may look like this:

```
Subject: I am on vacation
I am on vacation until July 22.
Your mail regarding "$SUBJECT" will be read when I return.
If you have something urgent, please contact
Joe Jones (jones@f40).
 --John
```

This `.vacation.msg` file will include the subject of the message in the reply.

No message is sent if the "`To:`" or the "`Cc:`" line does not list the user to whom the original message was sent or one of a number of aliases for them, if the initial "`From:`" line includes the string `–REQUEST@`, or if a "`Precedence: bulk`" or "`Precedence: junk`" line is included in the header.

OPTIONS

`–I` Initialize the `.vacation.pag` and `.vacation.dir` files and start `/usr/ucb/vacation`.

If the `–I` flag is not specified, and a *username* argument is given, `/usr/ucb/vacation` reads the first line from standard input (for a "`From:`" line, no colon). If absent, it produces an error message. The following options may be specified:

`–a`*alias*
: Indicate that *alias* is one of the valid aliases for the user running `/usr/ucb/vacation`, so that mail addressed to that alias generates a reply.

`–j` Do not check whether the recipient appears in the "`To:`" or the "`Cc:`" line.

`–t`*N* Change the interval between repeat replies to the same sender. The default is 1 week. A trailing `s`, `m`, `h`, `d`, or `w` scales *N* to seconds, minutes, hours, days, or weeks respectively.

USAGE

To start `/usr/ucb/vacation`, create a `.forward` file in your home directory containing a line of the form:

`\`*username*`, "|/usr/ucb/vacation `*username*`"`

where *username* is your login name.

Then type in the command:

```
/usr/ucb/vacation –I
```

To stop `/usr/ucb/vacation`, remove the `.forward` file, or move it to a new name.

If `/usr/ucb/vacation` is run with no arguments, it will permit you to interactively turn `/usr/ucb/vacation` on or off. It will create a `.vacation.msg` file for you, or edit an existing one, using the editor specified by the `VISUAL` or `EDITOR` environment variable, or `vi(1)` if neither of those environment variables are set. If a `.forward` file is present in your home directory, it will ask whether you want to remove it and turn off `/usr/ucb/vacation`. If it is not present in your home directory, it creates it for you, and automatically performs a `/usr/ucb/vacation -I` function, turning on `/usr/ucb/vacation`.

FILES

`~/.forward`

`~/.vacation.msg`

A list of senders is kept in the files `.vacation.pag` and `.vacation.dir` in your home directory.

SEE ALSO

`vi(1)`, `sendmail(1M)`.

User's Reference Manual

vi(1)

NAME

`vi, view, vedit -` display editor (visual)

SYNOPSIS

`vi` [*option ...*] [*file ...*]

`view` [*option ...*] [*file ...*]

`vedit` [*option ...*] [*file ...*]

DESCRIPTION

Layout

The description of `vi` is divided into the following parts:

- Options
- Command overview
- Introduction
- `vi` modes
  - Command mode
  - Input mode
  - Last line mode
  - `ex` command mode
  - `ex` input mode
- Screen layout
- Functionality
  - Saving the editor buffer and quitting `vi`
  - Text buffers
  - Movement commands
- Commands
  - Definitions
  - Control commands
  - Commands of the `vi` command mode
- Adapting `vi` to the terminal
  - Customizing the `vi` environment
- Files
- Environment variables
- Locale
- See also

`vi` is a display-oriented text editor. When invoked, the `vi` editor comes up in the `vi` command mode. If you have switched to `vi` input mode, pressing ESCAPE (or DEL) will take the editor back to the `vi` command mode.

OPTIONS

`–t` *tag*

The file containing the *tag* is loaded for editing. The editor positions the line containing the definition of

the tag in the center of the screen. A tags file containing the search strings for the definition must be in the same directory. This option is typically used by C programmers who want the editor to be positioned on the definition of a function or macro as soon as the file is opened. The *tags* file needed for this purpose must previously have been created with the ctags(1) command.

−l The Lisp mode is activated, and the text is correspondingly indented to comply with typical Lisp programming conventions.

The following vi commands have special meanings in Lisp mode: (,), {, }, [[and]].

−r *file*

Restores a vi session if the vi editor or the system crashed during your previous session.

Changes that are only in the editor buffer are not copied to the edited file after an editor or system crash. The UNIX operating system does, however, try to back up the buffer contents by creating a copy of the buffer, if possible. The *file* containing changes made prior to the crash is recovered and retrieved into the vi buffer. You can now continue editing the file or write the changes to another file.

−L Lists the names of all the files saved following a system or editor crash.

−R Opens the named *file* in read-only mode. This mode prevents the original *file* from being accidentally overwritten.

Caution:

The file can still be overwritten with the ex command w!.

−w*n* Sets the window size to *n* text lines.

−wn not specified:

The window size is set to the value of the LINES environment variable. If this variable has not been defined, the default value is as set in the terminfo definition for the terminal.

−c *command*

This option enables you to invoke vi and directly execute an ex command or use a vi movement command to position the editor in a specific line of the file being edited. If you use a movement command, the line required will be displayed in the center of the screen. By contrast, if you execute an ex command, the editor will be positioned in the last line of the file after the command is executed, unless the ex command you specify (e.g. search) positions the editor itself.

command not specified:

vi issues a usage message and aborts.

command specified:

command may be specified as a line number (*n*), a search command (/*pattern*, see the section entitled **Commands of the vicommand mode**), or any other ex command ("ex *command*" or 'ex *command*'). The specified command is executed when vi is loaded.

n *n* is an integer. vi is positioned in the *n*th line of the file.

/*pattern*

vi is positioned in the line containing *pattern*. *pattern* is a simple regular expression [see expressions(5)]. If the specified *pattern* includes metacharacters, you must escape such characters with a backslash \ or enclose *pattern* in single quotes '*pattern*', so that the shell does not interpret the metacharacters.

'*ex-command*'
"*ex-command*"

ex-command can be any arbitrary ex command. The ex command must be quoted (using single or double quotes) in order to prevent it from being interpreted by the shell. If the editor is not already positioned as a result of an ex command, vi will position to the last line of the file.

Example:

When vi is called with:

```
vi -c "set number showmode" appointments
```

the `appointments` file is opened, the cursor is positioned on the last line of the file and the line numbers and `vi` mode are displayed on the screen (see example in `ex(1)` description).

`-x` Encryption option: When invoked, `vi` executes the `ex` command `X` and prompts for a key. This key is then used to encrypt and decrypt text using the algorithm of the `crypt` command. `vi` checks whether or not the text to be read is encrypted. The temporary buffer file is likewise encrypted, using a transformed version of the key you enter here [see `crypt(1)`].

`-C` Encryption option: The `C` command has the same effect as the `X` command except that all text to be read is assumed to have been encrypted [see `crypt(1)`].

file Name of the file you want to edit. If you give a number of files, `vi` begins with the first file specified. The `ex` command `n` can be used to switch to the next file (see also `ex` commands `rew` and `ar`). If you invoke `vi` without a file name, an empty buffer is initially provided. If required, the contents of this buffer can be later written to a named *file* by using the `ex` command `w`*file*.

COMMAND OVERVIEW

The following overview summarizes the most common `vi` commands, grouping them together by function. In the section entitled **COMMANDS** you will find a detailed description of the commands in alphabetical order.

Invoking `vi`	
`vi` CR	Call `vi` with empty editor buffer
`vi` *file* CR	Call `vi` and load file

Quitting `vi`	
`ZZ`	Quit `vi`, writing editor buffer out to current file; as `:wq` CR
`:w` CR	Write editor buffer to current file
`:w` *file* CR	Write editor buffer to file
`:q` CR	Quit `vi` if changes saved
`:q!` CR	Quit `vi`, discarding unsaved changes
`:wq` CR	Quit `vi`, writing editor buffer out to current file; as `ZZ`

Positioning the cursor	
`h`	One character to left
`j`	One line down in same column
`+`	One line down and to start of line
`CR`	One line down and to start of line
`k`	One line up in same column
`-`	One line up and to start of line
`l`	One character to right
`H`	Top of screen
`M`	Middle of screen
`L`	Bottom of screen
`0`	Left edge of screen
|	Left edge of screen
`$`	Last character in line

^	First non-whitespace character in line
w	Start of next word
e	End of next word
b	Start of previous word
W	Start of next word (extended word)
E	End of next word (extended word)
B	Start of previous word (extended word)

Positioning the window	
CTRL-d	Half a screen down
CTRL-u	Half a screen up
CTRL-f	One screen down
CTRL-b	One screen up
CTRL-e	One line down
CTRL-y	One line up
z CR	Current line to top of screen
z.	Current line to middle of screen
z–	Current line to bottom of screen
1G	Start of file
nG	Line *n* to middle of screen
G	End of file

Deleting and restoring text	
x	Delete character at cursor position
X	Delete character to left of cursor
dd	Delete current line
dG	Delete from current line to end of file
dH	Delete from start of file to current line inclusive
D	Delete rest of line from cursor position
u	Restore deleted text from standard buffer
U	Restore current line
p	Write buffer contents after current cursor position
P	Write buffer contents before current cursor position

Changing text	
r	Replace current character with next character entered
R	Overwrite text starting at current character
o	Insert blank line below current line
O	Insert blank line above current line

a	Append text after current character
A	Append text to end of line
i	Insert text before current cursor position
I	Insert text at start of line

Copying	
Y	Yank current line
yy	Yank current line
Yp	Duplicate current line
yyp	Duplicate current line
:10,13t 20	Copy lines 10 through 13 below line 21
:.t.	Duplicate current line
:t.	Duplicate current line
:r *file*	Copy contents of *file* below current line

Cut and paste	
3dd	Delete 3 lines including current line
p	Insert saved lines
"a3dd	Delete 3 lines and save in buffer a
"ap	Insert saved line below current line

Search and replace	
/*xyz*	Search forward for *xyz*
/*xyz*<blank><blank>/	Search for *xyz* and 2 blanks
:s/*xyz*/*abc*/	Replace *xyz* with *abc* in current line
:/*old*/s/*old*/*new*	Find *old*, then replace *old* with *new*
:1,$s/*old*/*new*/g	Globally replace *old* with *new*

INTRODUCTION

vi displays a maximum of 23 lines of the file being edited. You can look at different parts of the file by moving this window. vi offers a wide range of text editing options. You can

- create, modify, copy and delete text
- position the cursor using simple commands (e.g. at the beginning or end of a word, line, sentence, paragraph, or the file)
- search and replace text patterns using regular expressions
- call a subshell
- process a range of the current file with a UNIX command
- edit multiple files during the same session and copy text from one file to another
- recover a file after an interrupted session.

vi is an enhanced version of the ex line editor [see ex(1)]. You can switch between the two editors while editing and you can execute ex commands from within vi.

`view` behaves like `vi`, except that *file* is opened in read-only mode and may not be changed.

`vedit` is a version of `vi` specially adapted for beginners. `vedit` behaves like `vi` with the options `report=1`, `showmode`, `nomagic` and `novice` set [on options refer to `ex(1)`].

vi modes

The `vi` editor provides you with a number of modes in which you can operate:

- `vi` command mode
- `vi` input mode
- `vi` last line mode
- `ex` command mode
- `ex` input mode

When invoked, the editor always comes up in the `vi` command mode, from which you can switch to other modes if required. The mode of the editor determines how keystrokes are interpreted.

vi command mode

When `vi` is invoked, it comes up in the `vi` command mode. No text can be entered in this mode. Keystrokes are immediately interpreted as `vi` commands, without being displayed on the screen. If the entered command is legal, it is executed, and the result is immediately displayed on the screen.

vi input mode

The input mode allows you to enter or modify text in the editor buffer. The editor switches to this mode when you enter one of the `vi` commands `A`, `a`, `C`, `c`, `I`, `i`, `O`, `o`, `S`, `s` or `R` (see the section entitled `vi` **commands**, **Commands of the** `vi` **command mode**). All following input characters, including various non-printing characters, are written to the buffer and the screen.

Most `vi` commands cannot be used in input mode. There are, however, a few `vi` commands that have a special significance in input mode. These include:

ESCAPE
: Exit input mode; return to `vi` command mode.

DEL
: Terminate input mode; return to `vi` command mode.

CR
: Newline.

CTRL-`@`
: Repeat last text input. You enter this command immediately after switching to input mode. `vi` inserts the last text entered in input mode provided this text consists of no more than 128 characters. Control is immediately switched back to command mode.

CTRL-`d`
: If you have defined automatic indentation, you can use CTRL-`d` to move the margin one tab stop to the left. You can set the shift width with the `shiftwidth` option. This command is only relevant at the beginning of a new input line and with the `autoindent` option set.

CTRL-`h`
: Go back one character.

CTRL-`t`
: One tab to the right in accordance with the value of `shiftwidth`. This command may only be entered at the start of a new input line.

CTRL-`v`
: Insert a non-printing character.

CTRL-`w`
: Go back one word.

\
: Escape for rubout character BACKSPACE.

vi last line mode

In the vi last line mode, you can enter vi, ex, and UNIX commands in the status line. The last line mode is activated by the input of a colon (:), slash (/), or question mark (?). Another method is to enter an exclamation mark (!), followed by a cursor movement command to a line. Commands entered in the last line mode must be terminated with CR or ESCAPE. The input of a command can be aborted by pressing DEL.

Caution:

Entering ESCAPE in the vi last line mode does not delete a command just entered, but causes it to be executed instead. In vi last line mode you must use DEL to delete an entered command.

`:` Entering a colon in the vi command mode causes all following input up to the next ESCAPE or CR to be interpreted as an ex command. (All ex commands are described under ex, ex **commands**.) After the colon is entered, the cursor moves to the status line at the bottom of the screen, and the colon as well as the input command are displayed there. Almost all ex commands can now be used, except for those that would switch ex to the ex input mode.

The percent character (%) represents the name of the file currently in the buffer; the hash character (#) stands for the name of the file last processed during the same session (see ex**, Current and alternate file**).

Example:

```
:! diff # % CR
```

vi writes the differences between the previously edited file and the current file to standard output. Neither of the two files is changed.

/ or ?

When a slash or question mark is entered, the cursor moves to the status line, allowing you to specify a regular expression for a forward search (with /) or a backward search (with ?). The vi command n repeats the search in the same direction; N repeats the search in the reverse direction.

`!` When an exclamation mark is entered in the vi command mode and a movement command specifying a line is given to indicate the scope, the cursor moves to the status line. The exclamation mark is then displayed at the start of the status line. If you enter a second exclamation mark as the movement command, the position is the current line. The movement command is not displayed. You can now enter a UNIX command, which must be terminated with CR or ESCAPE. The scope, i.e. the range from the current line to the specified position, becomes the input of the UNIX command (if it reads from standard input) and is replaced by its output (standard output and standard error). If no output is provided by the command (e.g. true), the specified range is deleted, i.e. replaced by a "null" string.

ex command mode

You switch to ex command mode by entering the command Q in vi command mode. You thus quit the display-oriented vi and switch to the line-oriented ex editor. Entering the command vi CR in ex command mode returns you to vi.

ex input mode

You switch to the ex input mode by entering the command a, i or c in the ex command mode. All input in this mode is written to the editor buffer. To terminate this mode you enter a single period in the first column of a line.

Most ex commands can be entered in one of two ways:

- from within vi command mode, by entering the vi command colon : followed by the ex command required.
- from within ex command mode.

SCREEN LAYOUT

A character screen on a terminal is generally made up of 24 lines and 80 columns. By default, the first 23 screen lines are used to display the text of the file being edited. In xterm sessions more than 24 lines can be

displayed. In this case the number of screen lines that vi uses is determined automatically.

The last line on the screen is a status and command line. The status line is used to enter commands and to display information and messages from the vi editor.

Text lines that do not fit on the screen are continued in the next line.

vi uses two special characters as markers in column 1 to identify the line status.

~ The tilde (~) identifies lines that are present on the screen but not in the editor buffer. These are lines past the end of the file. If the last text line of a file is located in the middle of the screen, for example, all lines thereafter will be marked with . The tilde disappears as soon as you enter text into the first such marked line. When you invoke vi with a new or empty file, all screen lines except for the first and last line (status line) are marked with this character.

@ The commercial at (@) marker indicates lines which vi cannot display correctly on the screen. There are two possible causes:

- A text line may extend over more than one line on the screen. This means that text lines may be present in the editor buffer, but the space on the screen after the last fully displayed text line does not suffice for another full line of text.
- Deleting individual lines from a file can mean that what appears on the screen no longer matches the contents of the editor buffer. In this case, the vi command CTRL-L can be used to redraw the screen.

The last line on the screen is the status line. It is used:

- to enter vi commands that begin with /, ?, ! or :
- to display messages
- for the output of ex commands
- to display the mode (the command mode is not displayed)

Acoustic signal

An acoustic signal is sent to the terminal when

- you press the ESCAPE or DEL keys in command mode
- you use an illegal command
- vi receives a SIGINT signal

If the vi editor receives a SIGINT signal (DEL during text input, or during the input of a command on the bottom line, the input is terminated (or the command canceled) and the editor returns to the command mode. The receipt of a SIGINT signal in the command mode produces an acoustic signal.

FUNCTIONALITY

vi always works on a buffer, not on files. When you start a vi session, a copy of the file you are editing is read into the editor buffer. If you omit the file name or specify a name that does not exist, you will begin with an empty buffer. All your editing changes will initially be recorded in this buffer; the original file is not changed until you copy the buffer contents back to it. This is done by explicitly saving the buffer during the session (with the ex command w) or while quitting the vi editor (with the ex commands wq, x, or the vi command ZZ). Use the ex command q! if you prefer to quit vi without modifying the original file. If you are creating a new file, it will not be created until you explicitly save the buffer contents to this file.

Saving the editor buffer and quitting vi

vi must be in vi command mode in order to save the editor buffer or quit vi.

You save the contents of your editor buffer to the edited file or to a specified file by means of the following ex command (see **Functionality of vi, Entering ex commands**):

:w [*file*]
(write, ex command) The contents of the editor buffer are saved to the specified file.

file not specified:

The contents of the editor buffer are saved to the edited file.

You have the following ways of quitting `vi`:

`:q` (quit, `ex` command) Quit `vi`. This command functions only if no changes have yet been made to the editor buffer or if the modified editor buffer was saved to a file.

`:q!` (quit, `ex` command) Quit `vi`. Changes made to the editor buffer are lost.

`:x` (exit, `ex` command) Quit `vi` and write the modified editor buffer to the current file.

`ZZ` (`vi` command) Quit `vi` and write the modified editor buffer to the current file.

`:wq` [*file*]
(write and quit, `ex` command) Quit `vi` and write the editor buffer to the specified *file*.

file not specified:

The contents of the editor buffer are written to the current file.

Text buffers

The `vi` editor makes use of

- one temporary buffer
- 9numbered buffers
- 26 named buffers

The contents of these buffers can be accessed by using `vi` commands in the `vi` command mode.

Temporary buffer

The last change that you make in the editor buffer is stored by `vi` in the temporary buffer. If, for example, you have deleted a line, the buffer will contain this line. It is used by the `vi` commands `U` and `u` (`undo`). The `u` command therefore allows you to reverse the effect of the last command that changed the editor buffer. `u` also reverses the effect of `u`.

If the last `vi` command was a copy (`y`), delete (`d`), or replace (except `R` and `r`) command, the temporary buffer can also be accessed by using the `vi` commands `P` and `p` (`put`).

`P` and `p` copy the contents of the temporary buffer before (`P`) or after (`p`) the current line. Unlike `U` (`u`), `P` (`p`) does not modify the temporary buffer. This allows you to use `P` (`p`) several times in order to insert the same text at multiple locations in the file.

Caution:

The contents of the temporary buffer are overwritten by each command that changes the editor buffer. The temporary buffer cannot therefore be used to copy text from one file to another.

The only commands that may be used between a copy, delete, or replace command and the `vi` command `P` (`p`) are cursor movement commands (see **Movement commands**). This is because other commands would change the contents of the temporary buffer.

If `vi` is in input mode and you use one of the arrow keys, the `vi` command `U` (`u`) cannot be subsequently used to undo editing changes.

Numbered buffers

When you delete one or more lines (`vi` command `d`), the text you delete is copied to the first of nine numbered buffers. If you now delete some more lines, the contents of buffer 1 are copied to buffer 2, and the most recently deleted lines are again preserved in buffer 1. Thus, buffer 1 always contains the most recently deleted material, buffer 2, the text from the previous delete, and so on, until buffer 9, which contains the material deleted nine steps ago.

The contents of numbered buffers can be retrieved using the `vi` commands `P` and `p`. Numbered buffers are referenced by a double quote and the number of the buffer in question. For example:

```
"4P
```

This `vi` command inserts the contents of buffer number 4 before the current line (or current position of the

cursor). The `vi` dot (.) command recovers the contents of the buffer from the next numbered buffer, provided your last `vi` command also retrieved text from a numbered buffer. In the above example, you would thus retrieve the contents of buffer number 5. The contents of numbered buffers are not lost when you switch files (with `ex` command `n` or `e`).

Named buffers

A copy (`y`) or delete (`d`) command can be used to write text into one of the twenty-six named buffers. These buffers are designated in lowercase from `a-z` or uppercase from `A-Z`. Lowercase and uppercase letters refer to the same buffer, however, with the following difference.

When you use a lowercase letter, the old buffer contents are overwritten by the new text, i.e. are deleted. By contrast, when uppercase letters are used, the old buffer contents are retained, and the new text is appended to the old. The `vi` editor does not change the contents of a named buffer during an editor session unless you explicitly overwrite its contents. `vi` writes text into a named buffer when you enter a double quote, followed by the buffer name and one of the `vi` commands `d` or `y`.

Examples:

`"A10dd`
: Delete (`dd`) the current line and 9 additional lines (10) and append them to the contents of buffer `A`.

`"by4w` Copy (`y`) the current and the next three words (`4w`) into buffer `b`.

`"cC`*string* ESCAPE
: Overwrite (`C`) the current line from the cursor position to the end of the line with *string*, and save the overwritten text in buffer `c`.

`"Ap` Recover the text from buffer `A` and insert (`p`) into the file after the current line or position.

Movement commands

`vi` provides you with a large number of commands, including search commands, which control the position of the cursor on the screen. These "movement" commands are entered in the `vi` command mode. The following overview lists the most important movement commands. The definitions of paragraph, section, sentence, word and extended word are given in the **COMMANDS, Definitions** section.

Character-oriented positioning

[*n*]CTRL-`h`
: *n* characters to the left

[*n*]`h` *n* characters to the left

[*n*]`l` *n* characters to the right

`0` to the start of a line

`^` to the first non-whitespace character in the line

[*n*]`|` to the first or *n*th column of the line

`$` to the last character in the line

[*n*]`f`*char*
: forward to the *n*th instance of *char*

[*n*]`F`*char*
: back to the *n*th instance of *char*

[*n*]`t`*char*
: forward to the position to the left of the *n*th instance of *char*

[*n*]`T`*char*
: back to the position to the right of the *n*th instance of *char*

Word-oriented positioning

[*n*]`b` back to the beginning of the *n*th previous word

[*n*]`B` back to the beginning of the *n*th previous extended word

[*n*]e forward to the end of the *n*th word

[*n*]E forward to the end of the *n*th extended word

[*n*]w forward to the beginning of the *n*th word

[*n*]W forward to the beginning of the *n*th extended word

Line-oriented positioning

[*n*]k up *n* lines in the same column

[*n*]j down *n* lines in the same column

[*n*]CTRL-J
down *n* lines in the same column

[*n*]CR down *n* lines to the first non-whitespace character

[*n*]+ down *n* lines to the first non-whitespace character

[*n*]– up *n* lines to the first non-whitespace character

[*line*]G
to the specified *line* number or end of file (if no *line* is given)

[*n*]$ to the last character of the *n*th line

Positioning by sentences, paragraphs, and sections

[*n*](back to beginning of *n*th previous sentence

[*n*]) forward to beginning of *n*th sentence

[*n*]{ back to beginning of current paragraph

[*n*]} forward to beginning of next paragraph

[*n*][[back to *n*th previous section boundary

[*n*]]] forward to *n*th section boundary

% to the matching parenthesis or brace in C source programs.

Screen-oriented positioning

[*n*]H to the first character in the *n*th text line on the screen

M to the middle of the screen

[*n*]L to the first character in the last text line on the screen

[*line*]z+
current line or specified *line* to top of screen

[*line*]z.
current line or specified *line* to middle of screen

[*line*]z–
current line or specified *line* to bottom of screen

Searching for patterns and markers

/*pattern*
forward to *pattern*

?*pattern*
back to *pattern*

'*marker*
to marked line (beginning of line)

`*marker*
to *marker*.

COMMANDS

As described above, you can enter ex commands either in vi command mode or in ex command mode. Only

the vi commands are described below. ex commands are described under ex(1) .

Hitting ESCAPE or DEL in the vi command mode cancels a partial command. ESCAPE sounds an acoustic signal if the editor is not in input mode or if there is no partially entered command.

Caution:

Entering ESCAPE in the vi last line mode does not delete a command just entered, but causes it to be executed instead. In vi last line mode you must use the DEL key to delete an entered command.

Unless otherwise specified, the commands are valid only in vi command mode and have no special effect in input mode.

Definitions

Extended word
: The term "extended word" refers to any sequence of characters which does not include any blanks, tabs, or newline characters. See Word.

Paragraph
: A paragraph is defined by the value of the ex option paragraphs. Blank lines and lines in which a section begins are also treated as paragraph boundaries.

Section
: A section is defined by the value of the ex option sections. Lines that begin with the form-feed character (CTRL-L) or a left brace { are also treated as section boundaries.

If the ex option lisp has been set, any left parenthesis at the start of a line is also treated as a section boundary.

Sentence
: A sentence end is defined by

a period (.),

an exclamation mark (!), or

a question mark (?)

followed by at least *two* spaces (to avoid confusion with abbreviations) or a newline character. To allow for quotations, any right parentheses, right square brackets, or single or double quotes between the punctuation and the newline character or spaces are ignored.

String
: Any sequence of characters.

Word
: A word is either any sequence of alphanumeric characters or a string of special characters. A word is terminated by a blank, tab, newline character or the following word.

See Extended word.

Examples:

((!!++;-.
: is one word

me&you2
: are 3 words (me & you2), but one extended word.

Control commands

To enter a control command, you hold down CTRL and press the key for the command required. Many control commands can be prefixed by a positive integer which specifies a *count*. The square brackets in the description signify that the specification of a *count* is optional. You must not press CTRL while entering *count*. The specified *count* is not displayed, but influences the effect of the subsequent command. If you specify an incorrect value, you can cancel your input by pressing ESCAPE. The *count* is used to:

- indicate the range of text to be affected in a movement command, e.g.

  5 CTRL-U

  scrolls back the screen by five lines.

- specify a repetition factor, e.g.

  `4` CTRL-`B`

  scrolls four screen pages back.

Unless explicitly stated otherwise, the default value for *count* is one.

CTRL-`@`

In `vi` input mode:

If you enter this and nothing else, the last text input is repeated at the position at which you enter CTRL-`@`. Afterwards, control is automatically returned to command mode.

CTRL-`@` only functions for a maximum of 127 characters.

[*count*]CTRL-`b`

(backward) Scrolls backward by 21 screen lines to display the window above the current one. A *count* can be used to indicate the number of windows the editor should go back. If possible, two overlapping lines from the previous screen are displayed in the new window.

[*count*]CTRL-`d`

In `vi` command mode:

count not specified:

count = 11, or last *count* used with CTRL-`d` or CTRL-`u`

(down) Causes the screen to scroll forward by half a window. The *count* specifies the number of text lines to be scrolled, and will be remembered during your `vi` session for future CTRL-`D` and CTRL-`U` commands.

In `vi` input mode:

count not specified:

In input mode, this command causes the cursor to back up by `shiftwidth` spaces over the indentation provided by the ex option `autoindent` or CTRL-`t` (see `ex`, `ex`**options**). CTRL-`d` only functions in a line in which no characters have been entered as yet.

count can also be: ^ or `0`

^CTRL-`d`

Cancels automatic indentation for this line.

`0`CTRL-`d`

Terminates the automatic indentation feature.

[*count*]CTRL-`e`

Scrolls forward by the number of lines given in *count*, leaving the cursor where it is if possible.

[*count*]CTRL-`f`

(forward) Scrolls forward by 21 screen lines. A *count* may be used to indicate how many screen pages to scroll forward.

CTRL-`g`

Displays the following information in the status line:

- the current file name
- possibly a status entry such as `[Modified]` if the file has been changed since the last time it was saved with the `ex` command `w`
- the current line number
- the total number of all text lines in the editor buffer
- the percentage of the current line to the total length of the file in lines.

This command is equivalent to the `ex` command `f`.

[*count*]CTRL-`h`

(same as BACKSPACE)

In `vi` command mode:

The cursor is moved by *count* characters to the left but no further than the left margin (see also the `vi` command `h`).

In `vi` input mode:

A *count* cannot be specified.

The cursor is then moved by one character to the left but stops at the beginning of the text just entered.

[*count*]CTRL-`j`
(same as MENU and ARROW-DOWN)

Moves the cursor *count* lines down in the same column.

This command is equivalent to CTRL-`n` and `j`.

CTRL-`l`
Clears and redraws the current screen. This command can be used when messages or reports are displayed on the screen (e.g. as a result of a `write`).

[*count*]CTRL-`m`
(same as CR)

Moves the cursor to the first character in the next line that is not a blank or tab. A *count* specifies the number of lines to go forward.

[*count*]CTRL-`n`
This command is equivalent to CTRL-`j` and `j`.

[*count*]CTRL-`p`
Moves the cursor *count* lines up in the same column.

This command is equivalent to the `vi` command `k`.

CTRL-`r`
(refresh) The current screen is cleared and redrawn. Previously deleted lines marked with `@` are removed. See CTRL-`l`.

CTRL-`t`
In `vi` input mode:

(tabulator) Inserts white space, defined by the value specified in the `ex` option `shiftwidth` (see `ex`, `ex` **options** section). The white space will, however, not be "visible" until the `vi` input mode is terminated. The inserted space can only be bakked over by using the CTRL-`d` command.

[*count*]CTRL-`u`
count not specified:

count = 11, or last *count* used with CTRL-`d` or CTRL-`u`

(up) Scrolls up a half-window of text. A *count* can be specified to scroll a specific number of lines. The *count* is remembered during the `vi` session for future CTRL-`d` and CTRL-`u` commands.

CTRL-`v`
In `vi` input mode:

"Quotes" or "escapes" a special character so that it can be placed in the text (e.g. escape or control characters). For instance, the sequence CTRL-V□ESCAPE places the ESC character into the file.

CTRL-`w`
In `vi` input mode:

(word) Backs up to the start of the previous word, but cannot be used to move beyond the text just entered. The backed over words remain displayed on the screen but are deleted from the buffer and must be re-entered, if required.

[*count*]CTRL-`y`
Scrolls backward by *count* lines (towards the beginning of the file), leaving the cursor where it is, if

possible.

CTRL-[
: (equivalent to ESCAPE)

  In vi command mode:

  Cancels a partially entered command; sounds an acoustic signal if there is none.

  In vi last line mode:

  Terminates input of a command just entered and executes it.

  In vi input mode:

  Terminates vi input mode.

ESCAPE
: See CTRL-[.

Commands of the vi command mode

Most vi commands accept a preceding number as an argument. Unless explicitly stated otherwise, the default value is one. This number, or count, is shown within square brackets in the description, which indicates that the entry is optional. The specified count is not displayed, but influences the effect of the following command. If you specify an incorrect count, you can cancel your input by pressing ESCAPE.

The following cases must be differentiated:

[*count*]*command*
: Here, *count* specifies how many times the command is to be executed. *count* must be a positive integer. *count* cannot be zero since 0 is also a vi command.

  count not specified:

  vi assumes a value of 1 for *count*.

[*line*]z
: vi places the line with the specified *line* number at the top, bottom, or center of the screen (see detailed description of the vi command z).

  line not specified:

  vi places the current line at the top, center, or bottom of the screen.

[*column*]|
: Specifies the column to which you position the cursor.

The following vi commands can be followed by a movement command *movecmd* to indicate the range to be affected by the command:

c, d, y, <, >, !, and =.

The <, >, !, and = commands, which only process whole lines, affect all lines that are fully or partially included in the specified range.

The range processed by the c, d, and y commands extends from (and including) the current cursor position to the position specified by the movement command. It is thus significant whether the cursor position is set by line or by word. For example, if the cursor position is set by line, all lines (including the current line) are processed as complete lines. If the cursor position is set by word, the processed range extends till the position before or after the word, depending on whether the cursor was set to the beginning or end of the word.

[*count*]a
: (append) Switches to vi input mode, appending the entered text after the current cursor position. A *count* can be used to specify how many times the inserted text is to be replicated, but only if the inserted text is limited to one line.

[*count*]A
: (append) Switches to vi input mode, appending the entered text at the end of the current text line. If the inserted text is all on one line, a *count* can be used to specify how many times the inserted text is to be replicated, but only if the inserted text is limited to one line. This command is equivalent to $a.

[*count*]b
(back) Backs up to the beginning of a previous word, where *count* specifies how many words to back up.

[*count*]B
(back) Backs up to the beginning of a previous "extended" word, where *count* specifies how many words to move.

[*count*]c*movecmd*
(change) Overwrites, i.e. changes a range of text. This command must be followed by a movement command.

- If the position specified by *movecmd* is in the current line, this position is marked by a $ character, and the editor enters the vi input mode to replace the range between the cursor and the $ by the entered replacement text.
- If the position specified by *movecmd* is in a new text line, the behavior of c depends on whether the scope of the change is defined by line or not. If the scope is defined by line, all fully or partially affected lines are deleted. However, if *movecmd* defines the scope by word, only the affected range is deleted. In both cases, vi switches to input mode to replace the deleted text by the new text entered. vi saves the deleted text in the temporary buffer, which means that it can be recovered with P or "1P, for example.
- A count can be passed through to the movement command.

[*count*]cc
The command *count*cc causes the whole of the current line (or *count* lines) to be changed, regardless of the cursor position in the current line.

[*count*]C
(change) Changes the rest of the current line. This command is equivalent to *count*c$.

[*count*]d*movecmd*
[*count*]dd
(delete) Deletes a range of text.

This command must be followed by a movement command. d deletes the range from the current cursor position to the end of the indicated range. If more than part of a single line is affected, the deleted text is saved in the numbered buffers 1 - 9, which means that it can be recovered with p.

count A count can be passed through to the movement command which indicates the range.

d The *count*dd deletes the whole of the current line (or *count* lines).

"*buffer*[*count*]d*movecmd*
"*buffer*[*count*]dd
Deletes and stores in named buffers.

The text from the current cursor position to the end of the indicated range is deleted from the file and stored in a named buffer. *buffer* is a single uppercase or lowercase letter.

D (delete) Deletes the rest of the current line. This command is equivalent to d$.

[*count*]e
(end) Moves the cursor forward to the end of the *count*th word.

[*count*]E
(end) Moves the cursor forward to the end of the *count*th extended word.

[*count*]f*char*
(find) f must be followed by a single character. vi scans the rest of the current line (from left to right) for that character and moves the cursor to it if found. A *count* is equivalent to repeating the search that many times.

A semicolon repeats the search in the same direction, while a comma repeats it in the reverse direction.

[*count*]F*char*

(find) F must be followed by a single character. vi scans backward in the current line (i.e. from right to left) for that character and moves the cursor to it if found. A *count* is equivalent to repeating the search that many times.

A semicolon repeats the search in the same direction, while a comma repeats it in the reverse direction.

[*line*]G

(go to) vi moves the cursor to the beginning of the line specified by line number *line*.

line not specified:

The cursor position is set to the last line of the file.

[*count*]h

(home) Moves the cursor by *count* characters to the left, but only in the current line.

[*count*]H

(home) Moves the cursor to the top line of the screen. If a *count* is given, the cursor is moved to that line on the screen, counting from the top. In both cases, the cursor is placed on the first non-whitespace character on the line.

[*count*]i

(insert) Switches to vi input mode, inserting the entered text before the current cursor position. If the new text is limited to one line, a *count* can be specified to indicate how many times the text is to be replicated.

[*count*]I

(insert) Switches to vi input mode, inserting the entered text before the first character of the current line. If specified, a *count* repeats the effect (see above).

[*count*]j

Moves the cursor down by *count* text lines in the same column.

This command is equivalent to CTRL-j and CTRL-n.

[*count*]J

Joins the current line with the next one, supplying appropriate white space: one space between words, two spaces after a period, and no spaces at all if the first character of the next line is a right parenthesis). A *count* can be used to specify the number of lines to be joined.

[*count*]k

Moves the cursor up by *count* text lines in the same column.

This command is equivalent to CTRL-P.

[*count*]l

Moves the cursor by *count* characters to the right, but only in the current line.

This command is equivalent to <space>.

[*count*]L

(last) Moves the cursor to the first non-whitespace character of the last text line on the screen. If specified, a *count* moves to that line counting from the bottom.

Example:

d3L deletes all lines from (and including) the current line up to (and including) the third text line from the bottom.

m*marker*

(mark) Marks the current cursor position. m must be followed by a *marker*, which can be any single lowercase letter.

Addressing

`*marker*

A backquote moves the cursor to the exact position of *marker*.

'*marker*
A single quote moves the cursor to the first character in the line containing *marker*.

`M` (middle) Moves the cursor to the middle line on the screen, at the first non-whitespace position on the line.

`n` Repeats the last / or ? scanning command.

`N` Repeats the last / or ? scanning command, but in reverse direction.

`o` (open) Switches to `vi` input mode and opens (i.e. inserts) a line below the current line.

`O` (open) Switches to `vi` input mode and opens a new line above the current line.

`p` Fetch text from temporary buffer.

(put) Writes text from the temporary buffer after the cursor. If the text was saved by `vi` commands as a whole line (e.g. `dd`, `Y`), it is inserted in a line below the current cursor position.

"*buffer*`p`
(put) Fetch text from numbered or named buffer

You can specify the *buffer* as:

- adigit from 1-9 to retrieve text from the buffer with the corresponding number
- aletter, to retrieve text from the named buffer with the corresponding name

`P`
"*buffer*`P`
(put) Similar to `p`, but putting the text before (or above) the current cursor position.

`Q` (quit) Quits `vi` mode and enters `ex` command mode. You must enter the `vi` CR command to return to the `vi` command mode.

[*count*]`r`*char*
(replace) Must be followed by a single character *char*; the character under the cursor is replaced by the specified one. If a count is specified, the next *count* characters are replaced with the single character given.

`R` (replace) Replaces several characters; `vi` enters the replace (or typeover) mode at the current cursor position. The characters on the screen are overwritten by the characters entered, until the input is terminated with ESCAPE.

[*count*]`s`
(substitute) Deletes the single character under the cursor and enters `vi` input mode; the entered text replaces the deleted character. A count specifies how many characters from the current line are changed. The last character to be changed is marked with a `$`, as in `c`. Example:

`5sblah blah blah` ESCAPE

replaces the character under the cursor and the next four characters to its right by the string `blah blah blah`. The same effect can be achieved with:

`c5lblah blah blah` ESCAPE

[*count*]`S`
(substitute) Switches to `vi` command mode. Whole lines are substituted (same as `cc`). A *count* can be used to indicate how many lines to replace.

[*count*]`t`
(to) *t* must be followed by a single character. `vi` scans forward through the remainder of the current line for the specified character. If it finds the character, it places the cursor on the character which precedes it. A count is equivalent to searching for the *count*th occurrence of the character. Using , or ; to repeat the search is not meaningful.

[*count*]`T`*char*
(to) `T` must be followed by a single character. `vi` scans backwards through the current line for the specified character. If it finds the character, it places the cursor on the character which comes after it. A count is equivalent to searching for the *count*th occurrence of the character. Using , or ; to repeat the

search is not meaningful.

`u` (undo) Reverses the last change made to the temporary buffer. If repeated, will alternate between these two states, thus is its own inverse.

Example:

`d3w` Deletes three words from the current cursor position onward.

`u` Puts back the three words.

`u` Deletes the three words again.

The contents of the temporary buffer can be accessed by using the `p` and `P` commands.

If `u` is used after an `insert` command that inserted at least one new text line, the text deleted with `u` is saved in numbered buffer 1.

`U` (undo) Restores the current line to its state before the cursor was last moved to it. `U` only functions so long as you have not left the current line.

[*count*]`w`

(word) Moves forward to the beginning of the next word; a *count* specifies how many words to move.

[*count*]`W`

(word) Moves forward to the beginning of the next extended word; a *count* specifies how many words to move.

[*count*]`x`

Deletes the character at the current cursor position. With a *count*, deletes *count* characters forward from the current cursor position, but only on the current line.

[*count*]`X`

Deletes the character before the cursor. A *count*, if specified, deletes that many characters before the cursor on the current line.

[*count*]`y`*movecmd*
[*count*]`yy`

Saves text in the temporary buffer

(yank) This command must be followed by a movement command to indicate the range. The specified text range is copied (yanked) into the temporary temporary buffer.

count A*count* copies the specified range that many times.

`y` The `yy` command causes the whole of the current line to be yanked into the buffer.

Examples:

`yw` yanks the text from the cursor position to the end of the next word into the temporary buffer.

`4y3w` yanks 12 words into the temporary buffer.

`4yy` yanks the current text line and the three lines below it into the temporary buffer.

`"`*buffer*[*count*]`y`*movecmd*
`"`*buffer*[*count*]`yy`

Yank text into named buffers

(yank) If the `y` command is preceded by the name of an named buffer (`"`*buffer*), the text will also be copied into this buffer. *buffer* must be specified as an uppercase or lowercase letter.

Examples:

`"a4yy` Yank the current text line and the three lines below it into named buffer `a`.

`"ap` Put back the contents of `a` after the current cursor position.

[*count*]`Y`

Yank line(s) into temporary buffer.

(yank) Yanks a copy of the current line or *count* lines into the temporary buffer (same as `yy`).

"*buffer*[*count*]Y

Yank line(s) into named buffers.

(Y – yank) Yanks a copy of the current line (or *count* lines) into the named buffer given in *buffer*, where buffer is an uppercase or lowercase letter. This format corresponds to "*buffer*yy.

[*line*]z[*count*]*char*
/*pattern*/z[*count*]*char*

Moves the text line specified by *line* or *pattern* to the screen position defined by *char*.

char must follow z and may assume the following values:

CR or plus sign (+)
: Specifies the top of the screen

dot (.)
: Specifies the center of the screen

minus sign (–)
: Specifies the bottom of the screen

line indicates the line number of the text line to be positioned.

line not specified:

The current line is used by default.

pattern identifies the line in which *pattern* first occurs.

count is an integer between 1 and 23 that may be given after z to specify the lines to be redrawn.

count not specified:

count = 23.

ZZ Exits the vi editor, writing out the buffer contents to the file with the current file name (see ex command, f) if the buffer was changed since the last save (with the ex command w). This command is equivalent to the ex command x.

[*count*]<space>

The cursor is moved by *count* characters to the right (stops at the end of the line).

This command is equivalent to l.

[*count*]!*movecmd*

An ! with a movement command allows you to switch to the vi last line mode to process text lines with a UNIX command. The ! must be followed by a movement command to indicate the scope, which must go beyond the current line. The specified line range (from the current position to the one given by the movement command) is then passed as standard input of the UNIX command and is replaced by the output (standard output and standard error) of the system command. Lines are always passed as complete text lines.

The cursor does not move to the status line until the movement command has been entered. The UNIX command to be executed must be terminated with the CR key.

movecmd
: Movement command to a particular line. Here are some examples:

  3j move down 3 lines

  *line*G go to the line number given in *line*

  L move to the last screen line.

  A second ! can also be specified for *movecmd*; this passes the current line as standard input to the UNIX command. The second exclamation mark can be preceded by a count to indicate the number of lines to be passed to the UNIX command, starting with the current line.

count Serves as a repetition factor.

Examples:

!!wc CR

Passes the current line to `wc` as input and replaces it by the output of `wc`.

`!1Gsort` CR
Sorts the range from the current line to the start of the file.

`3!$tr a b` CR
Replaces all occurrences of `a` by `b` in the entire current line and the two following lines.

[*count*]`$`
Moves the cursor to the end of the current line. A *count* specifies the number of lines to go forward (e.g. `2$` moves the cursor to the end of the next line).

`"` Precedes the name of a named or numbered buffer. See the `vi` commands `d`, `p`, and `y`.

`%` Moves the cursor to the parenthesis, brace, or bracket that matches the one at the current position. Left and right parentheses `( )`, square brackets `[ ]`, and braces { } represent matching pairs.

`&` Repeats the last `ex` command `s`; is equivalent to the `ex` command `&`.

'*marker*
(single quote)
Move cursor to marked line.
'*marker* moves the cursor to first non-whitespace character in the marked line. *marker* is a lowercase letter that was set as a marker with the `vi` command `m`.
If the 'command is used as a movement command for some other command, all lines in the specified range are treated as whole lines, including the current and the marked line.
Example:
`d'a`
Deletes all lines from the current line up to the marked line, including both extremes.

`''` (two single quotes)
Move cursor to previous current line.
The "command moves the cursor to the beginning of the line in which it was located prior to the execution of one of the `vi` commands below:
`H`, `L`, `M`, `n`, `N`, `%`, `'`, `(`, `)`, `[[`, `]]`, `` ` ``, `{`, `}`, */pattern*□CR, `?`*pattern*□CR.
If the "command is used as a movement command for some other command, all lines in the specified range are treated as whole lines, including the current and the marked line.
Example:
`d''`
Deletes the range from the current line to the previous current line. You should first use "to verify the location of the previous current line. If none exists, `vi` deletes the range from the current line to the beginning of the file. The deleted text can be retrieved with `u`.

[*count*]`(`
Moves the cursor back to the beginning of a sentence; a *count* moves back that many sentences.
If the `ex` option `lisp` is set, moves to the beginning of a `lisp` `s`-expression (see `ex`, **ex options**). Sentences also begin at paragraph and section boundaries (see { and `[[`).

[*count*]`)`
Moves forward to the beginning of the *count*th sentence.

[*count*]`+`
Moves the cursor to the first non-whitespace character in the next line. Use a *count* to move more than a line (as in CTRL-`m`).

[*count*]`,`
Reverses the last `vi` search command (for a single character):
`f`, `F`, `t`, or `T`,

searching in the opposite direction in the current line. A *count* can be used as a repetition factor.

[*count*]–
(hyphen) Moves the cursor backward to the first non-whitespace character in the preceding line. A *count* specifies how many lines to move back.

[*count*]**.**
(dot) Repeats *count* times the last `vi` command which changed the buffer.

If the last command given was used to output the contents of a numbered buffer, a following dot command retrieves the contents of the next numbered buffer in sequence.

/pattern CR
The / immediately takes you to the command line (`vi` last line mode) where you can specify a regular expression for *pattern*. The search begins when you terminate *pattern* with CR. `vi` then scans forward for the next occurrence of a matching string and moves the cursor to it. If no string matching *pattern* is found, the cursor remains in the current position. The interrupt signal `SIGINT` can be given to terminate the search.

Since the `ex` option `wrapscan` (abbreviated to `ws`) is set by default, the search wraps around at the end of the file and continues downward from the start of the file to the current cursor position. The `ex` command `set nows` causes the search to terminate at the end of the file (see the `ex` command `set` and the section on "ex options" under `ex`).

The `vi` commands `n` and `N` repeat the search in the same and reverse directions, respectively.

If the / is used as a movement command with some other command to specify an extent of text, the defined region is from (and including) the current cursor position to just before the matched string.

Example:

If a line contains the following text:

```
Larry Speakes, the Speaker of the House, spoke well.
```

And the cursor is positioned at the comma after Speakes, entering the command:

`d/ spoke` CR

will leave:

```
Larry Speakes spoke well.
```

`?` Scans backwards; the reverse of `/` (see `/`).

`0` Moves the cursor to the first character on the current line. The zero is not interpreted as a `vi` command when preceded by a non-zero digit.

`:` The `vi` command colon (`:`) switches to `vi` last line mode, allowing you to enter an `ex` command. The `ex` command is terminated by entering CR.

If the `ex` command you wish to enter is longer than one input line, you will need to switch to the `ex` command mode with `Q`.

[*count*]*;*
Repeats the previous `vi` search command (for a single character):

`f`, `F`, `t`, or `T`.

A *count* can be used for a number of repetitions.

[*count*]<*movecmd*
Must be followed by a movement command to another line; shifts the inclusive range from the current to the specified line by `shiftwidth` columns to the left. See section on `ex` **options** under `ex`. A *count* is passed through to the movement command.

A second < character can also be specified as *movecmd*; << causes the current line to be shifted to the left (or *count* lines, including the current line).

[*count*]=*movecmd*
Must be followed by a movement command to another line. If the `ex` option `lisp` is set, the specified

lines are re-indented as if they had been typed in with the `autoindent` option set. A *count* can be used to specify how many lines are to be processed. *movecmd* may also be a second equals sign, which causes only the current line (or *count* lines) to be indented.

[*count*]>*movecmd*
This `vi` command shifts specified lines by `shiftwidth` characters to the right (see <).

`*marker*
(backquote)
Move cursor to set marker
marker is a lowercase letter which identifies a marker that was set with the `vi` command `m`. `*marker* places the cursor at the exact character position marked.
If the ` command is used as a movement command with some other command to specify its scope, the defined region extends from (and including) the current cursor position up to just before the previous current position.
Example:
```
d` a
```
Deletes the range from the current cursor position up to the marker `a`.

``
(double backquotes)
Move marker to previous current position
The `` command moves the cursor to the position it occupied before one of the `vi` commands listed below was executed:
`H`, `L`, `M`, `n`, `N`, `%`, `"`, `(`, `)`, `[[`, `]]`, ``, `{`, `}`, */pattern*□CR, ?*pattern*□CR.
If the `` command is used as a movement command with some other command to specify its scope, the defined region extends from (and including) the current cursor position up to just before the previous current position.

[*count*]`[[`
Backs up to the section boundary given by *count*.

[*count*]`]]`
Moves forward to the section boundary given by *count*.

^
Moves the cursor to the first non-whitespace character on the current line.

[*count*]{
Moves back to the beginning of the current paragraph. You can use a *count* to indicate how many paragraphs to move.

[*count*]}
Moves forward to the beginning of the next paragraph. A *count* can be used to indicate how many paragraphs to move.

[column]|
Moves the cursor to the specified *column*, if possible.
column not specified: *column* = 1

~
Case shifting; converts the character at the current cursor position from uppercase to lowercase or vice versa as appropriate, and moves the cursor one character to the right.

ADAPTING VI TO THE TERMINAL

The output of the `vi` program depends on the type of terminal used. The `vi` editor need not be adapted in the UNIX environment.

`vi` uses the value of the environment variable `TERM` to locate the required terminal information in a database (`/usr/lib/terminfo/*/*`). If `TERM` has not been defined or if you are working at some other type of terminal, you must reset the value of `TERM`. You can do this in a number of ways:

- set and export the shell variable `TERM`.

- define and export TERM in your .profile.
- use the ex command set term (in ex command mode only).
- enter the ex command set term in the .exrc file.
- define and export the EXINIT environment variable in your .profile.
- if all else fails, consult your system administrator.
- Make sure you only specify the actual name of your terminal for TERM. The use of other names may cause your terminal to malfunction.

Customizing the vi environment

Default values are set using ex options. The available options are described under ex, section ex **options**.

The ex command set all displays a list of all the current values. For example:

directory=/tmp
: Directory for the editor buffer

nonumber
: Line numbers are not displayed

report=5
: A message is issued when more than 5 lines are changed

scroll=11
: The screen is scrolled by 11 lines when CTRL-D is entered

noshowmode
: The vi mode is not displayed

term=97801
: Terminal type

window=23
: Number of screen lines for the text window

wrapmargin=0
: No automatic wraparound

There are two types of option, those with Boolean values and those with non-Boolean values. showmode is an example of a Boolean type option. If it is set, set all displays the string showmode. If it is not set, set all displays noshowmode. With non-Boolean types, set all displays the option name and the value to which it is set. scroll is an example of a non-Boolean option.

Note: If you have trouble with the arrow keys in vi (for example, if pressing an arrow key switches you from input to command mode), you should switch off the timeout option (set notimeout).

The vi environment can be customized to suit your requirements and working habits. For example, vi can be made to display line numbers of the current mode of operation in the status line.

There are three ways of changing the default values for vi:

- during a vi session
- by using the EXINIT environment variable
- by using the .exrc file

Let us assume, for example, that you want to have the line numbers and the vi mode displayed.

- During a vi session:

  To display the line numbers, use the ex command set number during a vi session. The ex command set showmode causes vi to indicate when it is in input mode. The customized environment will only be effective for the duration of the session.

- By using the EXINIT environment variable:

  You must write the following lines into the .profile file in your HOME directory:

  ```
  EXINIT='set number showmode'
  ```

□
□
export EXINIT

Using the command . .profile you make known the new variable to your shell. If you now call vi, line numbers will be displayed, and vi will inform you when you are in the input mode. This default setting is always effective.

- By using the .exrc file:

  You must create this file yourself. The .exrc file can be used to define permanent vi settings. To ensure that vi displays line numbers and the mode, you must enter the following line into this file:

  set number showmode

  The .exrc file has to be in your HOME directory. For security reasons, vi does not evaluate .exrc files in subdirectories. The .exrc in the HOME directory is always read.

The default values defined for vi are also valid for ex.

It is possible to combine the various methods, but on certain conditions. Values set within a vi session have the highest priority. You may therefore change the current values at any time during a session by means of the ex command set.

If EXINIT is set, the $HOME/.exrc file is not executed.

ENVIRONMENT VARIABLES

LINES Number of lines per screen.

TERM The type of terminal used.

EXINIT
: Environment variable with default values for ex and vi.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

FILES

$HOME/.exrc
: File containing default values for ex and vi. These default values are overridden by values set using EXINIT or during a vi session.

SEE ALSO

edit(1), ex(1).

el Lozy, M.: *Editing in a UNIX Environment. The vi/ex Editor*.

w(1-ucb) - zcat(1)

User's Reference Manual

w(1)
BSD Compatibility Package

NAME

w - who is logged in, and what are they doing

SYNOPSIS

/usr/ucb/w [–hls] [*user*]

DESCRIPTION

The w command displays a summary of the current activity on the system, including what each user is doing. The heading line shows the current time of day, how long the system has been up, the number of users logged into the system, and the load averages. The load average numbers give the number of jobs in the run queue averaged over 1, 5, and 15 minutes [cf. sar(1) , –q option and mpstat(1M)].

The 7 fields displayed after the heading line are: 1) the user's login name, 2) the name of the tty the user is on, 3) the time of day the user logged on (in *hours*:*minutes*), 4) the idle time – that is, the number of minutes since the user last typed anything (in *hours*:*minutes*), 5) the CPU time used by all processes and their children on that terminal (in *minutes*:*seconds*), 6) the CPU time used by the currently active processes (in *minutes*:*seconds*), 7) the name and arguments of the current process.

If a *user* name is included, output is restricted to that user.

OPTIONS

–h Suppress the heading.

–l Produce a long form of output, which is the default.

–s Produce a short form of output. In the short form, the tty is abbreviated, the login time and CPU times are left off, as are the arguments to commands.

EXAMPLES

```
$ w
  3:38pm  up 10 days,  8:03,  6 users,  load average:  0.00, 0.00, 0.00
User     tty           login@  idle   JCPU   PCPU  what
meer     pts/17         7:50am  7:46                -sh
cle      pts/2         11Jan95  3:59                -su
hannibal pts/1         Mon 4pm 26:48                rlogin ubahn
san      pts/45         1:21pm  2:16                -ksh
reinsch  pts/27         8:01am  7:36                -sh
hofu     pts/49         8:26am             1        ksh
```

The CPU time is only an estimate, in particular, if someone leaves a background process running after logging out, the person currently on that terminal is charged with the time.

Background processes are not shown, even though they account for much of the load on the system.

Sometimes processes, typically those in the background, are printed with null or garbaged arguments. In these cases, the name of the command is printed in parentheses.

w does not know about the conventions for detecting background jobs. It will sometimes find a background job instead of the right one.

FILES

/var/adm/utmp

/dev/kmem

/dev/drum

SEE ALSO

ps(1) , sar(1) , who(1) , mpstat(1M) , whodo(1M) , utmp(4) .

User's Reference Manual

wait(1)

NAME

`wait -` wait for background processes to terminate

SYNOPSIS

`wait` [*process_ID*]

DESCRIPTION

The shell built-in `wait` causes the shell to wait

- until a previously started background process is complete (if you call `wait` with the required process ID), or
- until all previously started background processes terminate (if you call `wait` without arguments).

When a command is run in the background (with &), the interactive shell does not normally wait for it to finish before prompting you for a new command. By contrast, if a command is started as a foreground process, the shell will always wait for it.

This also applies with respect to shell scripts. Thus, if a script contains a command that is to process the output of a previously started background command, `wait` can be used to ensure that the background command terminates first.

If the error message

```
fork failed – too many processes
```

appears when you try to run a process in the background, you may be able to lessen the system load by running `wait`. If this has no effect, it may be that the system table is full or that there are too many active foreground processes.

Note:

If a pipeline has 3 or more stages, the shell groups processes into pairs and spawns a process to control each pair. Hence the original processes are not child processes of the shell. `wait` cannot wait for processes which are not children of the shell.

OPERANDS

process_ID

Process identification number of the background process whose completion is to be waited for by the shell.

In the Bourne shell only one process ID may be specified. If more than one process ID is given in the command line, `wait` will only consider the first one and will ignore the rest without issuing an error message.

The exit status returned by `wait` is that of the specified background process. If the process you specify is not active, `wait` behaves as it does if you do not specify a PID (see **EXIT STATUS**).

The shell always assigns the process ID of the last command run in the background to the `!` (exclamation point) variable. The contents of this variable can be accessed with `$!`.

process_ID not specified:

`wait` waits for all your shell's currently active background processes to complete and returns an exit status of 0.

EXIT STATUS

If `wait` was called with no arguments, the exit status is always 0.

If you call `wait` with a process ID, the exit status returned by `wait` is that of the background process selected with the given process ID.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Example 1

The shell is to wait for an active background process:

```
$ find / -name tst -print 2>/dev/null &
3456
$ wait 3456
home/martin/PASCAL/tst
...
```

Example 2

The shell script patience illustrates how to wait for a specific background command to terminate. The contents of this shell script are as follows:

```
: Invocation with sh patience
sort list > sorted &
pid1=$!
tar cvf /dev/dsk/f0t /home/anna &
pid2=$!
...
wait $pid1
pg sorted
```

The process IDs of the two background command are stored in the variables pid1 and pid2, respectively. These process IDs are thus available for future use, as the ! variable always holds the process ID of the last background command started.

The wait command causes the script to wait for the sort command to complete before displaying the contents of sorted with the pg command.

NOTES

Some differences in behavior may occur when using wait, depending on which shell is being used. The possible differences are not described specifically.

SEE ALSO

ksh(1), sh(1).

User's Reference Manual

wc(1)

NAME

wc - word, character and line count

SYNOPSIS

wc [*option ...*] [*file ...*]

DESCRIPTION

wc counts the number of lines, words and bytes, or characters contained in files and writes the results to standard output.

OPTIONS

No option specified:

wc outputs three numeric values for the number of lines, words, and bytes.

−c wc outputs the number of bytes. Must not be combined with −m.

−l wc reports the number of lines, based on the number of newline characters counted by wc.

−m wc outputs the number of characters, including blanks, tabs, and newline characters. Must not be combined with −c.

−w wc reports the number of words. A word is defined as a non-empty string delimited by whitespace characters. Whitespace characters are blanks, tabs and newline characters.

−− If *file* begins with a dash (−), the end of the command-line options must be marked with −−.

file Name of the file whose lines, words and bytes, or characters are to be counted. The file name will be printed along with the counts.

More than one file may be specified. If several files are named, wc prints an additional line with a grand total of the individual values for all specified files.

file not specified:

wc reads from standard input.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

LC_CTYPE defines which characters are treated as whitespace characters.

If LC_MESSAGES or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Listing line, word, and character counts for the files logic, plan and rem.

```
$ wc logic plan rem
   27    139    1077 logic
    5     15     140 plan
    3      6      51 rem
   35    160    1268 total
```

Example 2

Counting the number of files in the current directory.

```
$ ls | wc -l
     31
```

Example 3

Checking the number of users working on the system.

```
$ who | wc -l
      6
```

Example 4

Counting the number of unique words in a file.

```
$ cat file | sed 's/ */\
> /g' | sort -u | wc –l
```

Explanation:

`sed` creates a list of all words in `file` by replacing one or more blanks by newline characters. `sort -u` sorts this list, removing all repetitions. `wc -l` then counts the lines in this list and prints the total.

SEE ALSO

`cksum`(1) .

User's Reference Manual

wcheck(1)

NAME

`wcheck -` search for modified files (write check)

SYNOPSIS

`wcheck [ –a ]`

DESCRIPTION

`wcheck` lists all the files belonging to the user that have been modified between the last logout and the login for the current session.

When a user logs into the system for the very first time, the last logout is defined as January 1 1970, 1:00 am.

The `wcheck` command should be entered in the `.profile` file for security reasons. Users can then check each time they log in that their files have not been manipulated in the meantime.

If a file that has been modified or read in the specified timespan is changed or read again during the current session, the time stamp is renewed. `wcheck` then no longer recognizes this file as having been changed. `wcheck` should therefore be called immediately after system login.

OPTIONS

`–a` `wcheck` also lists those files that have been read between the last logout and the login for the current session.

FILES

`/var/adm/wtmpx`
: The `/var/adm/wtmpx` file contains user and administration information, including the login and logout times.

User's Reference Manual

whatis(1)
BSD Compatibility Package

NAME

`whatis` - display a one-line summary about a keyword

SYNOPSIS

`/usr/ucb/whatis` *command* ...

DESCRIPTION

`whatis` looks up a given *command* and displays the header line from the manual section. You can then run the `man`(1) command to get more information. If the line starts "*name*(*section*)□..." you can do "`man`□*section*□*name*" to get the documentation for it. Try "`whatis ed`" and then "`man 1 ed`" to get the manual page for `ed`(1).

`whatis` is actually just the `-f` option to the `man` command.

CAVEAT

This `whatis` command will not display information from Reliant UNIX man pages. It is included for users who have BSD man pages.

FILES

`/usr/share/man/whatis`
Database

SEE ALSO

`man`(1), `catman`(1M).

User's Reference Manual

which(1)
BSD Compatibility Package

NAME

`which` - locate a command; display its pathname or alias

SYNOPSIS

`/usr/ucb/which` [*filename*] ...

DESCRIPTION

`which` takes a list of names and looks for the files which would be executed had these names been given as commands. Each argument is expanded if it is aliased, and searched for along the user's path. Both aliases and path are taken from the user's `.cshrc` file.

DIAGNOSTICS

A diagnostic is given for names which are aliased to more than a single word, or if an executable file with the argument name was not found in the path.

NOTES

Only aliases and paths from `~/.cshrc` are used; importing from the current environment is not attempted.

`which` must be executed by `csh(1)`, since only `csh` knows about aliases. If you are using `sh` instead of `csh`, `whence -v` provides similar functionality.

To compensate for `~/.cshrc` files in which aliases depend upon the `prompt` variable being set, `which` sets this variable. If the `~/.cshrc` produces output or prompts for input when `prompt` is set, `which` may produce some strange results.

FILES

`~/.cshrc`
: Source of aliases and path values

SEE ALSO

`csh(1)`, `ksh(1)`, `sh(1)`.

who(1)

NAME

`who` - show who is on the system

SYNOPSIS

Format 1

`who` [*option* ...] [*file*]

Format 2

`who` `-q` [`-n` *number*] [`--`] [*file*]

Format 3

`who am i`

Format 4

`who am I`

DESCRIPTION

`who` provides information on:

- the login and terminal name under which you are currently working
- the login name, login time, and terminal name for each current system user
- the process ID of the command interpreter (shell) being used
- the last time a terminal was used
- when logins, logouts, and system breakdowns have taken place since the `/var/adm/wtmp` file was cleared by the system administrator.
- the last time the system clock was changed
- the processes spawned by the `init` process
- `who` normally gets its information from the `/var/adm/utmp` file. The information in this file is brought up to date every time a login is performed. In single-user mode no logins are performed. Thus after a shutdown to single-user mode `who` may not be able to supply accurate information about the current login status. Use `who am i` instead.

  To obtain the same information if the `/var` directory (or the directory corresponding to it in your system configuration) is not mounted in the file system in single-user mode, you should proceed as follows: Call the `who` command giving the temporary files `tmp_utmp` and `tmp_wtmp` as arguments. These files reside in the `/stand` directory.

OPTIONS

Format 1: List detailed information

`who` [*option* ...] [*file*]

No option specified:

`who` lists the following for every system user currently logged in:

- login name under which the user logged in
- name of the terminal on which the user logged in
- login time

The columns in the output are described below in **Output**.

`-a` (a – all) Turns on options `-b`, `-d`, `-l`, `-p`, `-r`, `-t`, `-T` and `-u`.

`-b` (b – boot) Shows the date and time of the last reboot.

`-d` (d – dead) Shows all processes that have terminated and not been respawned by `init`. Produces a

list of dead processes together with their exit status and the number of the signal which terminated the process. This may be useful in determining why a process died.

`–H` (H – headings) Prints headings above the output columns.

`–l` (l – login) Lists the processes where the system is waiting for someone to log in. In such cases the entry in the `NAME` field is the name of the program (or `LOGIN`). The other fields have their usual meanings.

`–m` `who` only outputs information on the active terminal.

`–p` (p – process) Shows all processes spawned by the `init` process. The `NAME` field shows the name of the program executed by `init` as listed in `/etc/inittab` [see inittab(4)].

The `LINE` field is irrelevant in conjunction with the `–p` option, so only a dot is output in this field. The `COMMENTS` field is not displayed unless you combine `–p` with `–a` or with `–l` and/or `–u`. In these cases the comment field contains the entry from the corresponding line in `/etc/inittab` [see inittab(4)].

`–r` (r –run level) Shows the current run level of the `init` process. The `LINE` field indicates the current run level, the `TIME` field the date it was entered.

`IDLE` shows the current run level in numeric form, `PID` shows how often the system has been in this state before, and `COMMENTS` indicates the system's previous run level. The `NAME` field has no meaning with the `–r` option.

`–s` (s – standard) Lists the users who are currently logged in. The `NAME` field gives the login name, `LINE` is the device name (without the `/dev/`) of the terminal on which the user logged in. `TIME` shows when the user logged in.

This option is the default. `–s` cannot be combined with `–a`, `–d`, or `–T`.

`–T` (T – terminal) The terminal state is output in addition to the default values. The `LINE` field shows whether other users can write to the terminal: a plus sign `+` indicates that they can, a minus sign `-` means that they cannot. The system administrator can write to any terminal. If it is not possible for the system to obtain this information, a question mark `?` appears in this field.

`–t` (t – time) Shows the last time the system administrator changed the system clock (using the `date` command).

`–u` (u – user) Lists those users who are currently logged in. The `NAME` field is the user's login name. `LINE` is the device name (without the `/dev/`) of the terminal on which the user logged in. `TIME` shows when the user logged in.

The `IDLE` field provides information on the last time there was activity on the terminal in question: a dot (.) indicates activity within the last minute. If the terminal has not been used for over 24 hours or since the system was last booted, the entry is marked `old`.

The `PID` field indicates the process ID of the command interpreter (shell) the user is working with. `COMMENTS` is a comment about the terminal as contained in `/etc/inittab` [see inittab(4)]. This comment may provide information about where the terminal is located and similar topics.

`––` If *file* begins with a dash (–), the end of the command line options must be marked with `––`.

file Name of the file from which `who` obtains its information. *file*, if given, will typically be `/var/adm/wtmp`. `who` then provides information about logins, logouts and system breakdowns that have taken place since this file was last cleared by the system administrator.

file not specified:

`who` obtains its information from the file `/var/adm/utmp`.

Output

The following section lists the headings of the columns output by `who` and discusses the information displayed in each column.

The output columns have different meanings in some cases with the `–b`, `–d`, `–p`, `–r`, and `–t` options.

NAME Indicates the user's login name.

LINE Gives the device name (without the /dev/) of the terminal on which the user logged in.

TIME Shows when the user logged in.

IDLE Provides information on the last time there was activity on the terminal in question: a dot (.) indicates activity within the last minute. If the terminal has not been used for over 24 hours or since the system was last booted, the entry is marked old.

PID Indicates the process ID of the command interpreter (shell) the user is working with.

COMMENTS
Contains a comment about the terminal as contained in /etc/inittab [see inittab(4)]. This comment may provide information about where the terminal is located and similar topics.

Format 2: List concise information

who –q [–n *number*] [––] [*file*]

–q (q – quick) who only outputs the names and number of users currently logged on. Additional options, with the exception of –n, are ignored.

–n *number*
who displays *number* users per line. *number* must be at least 1.

–– If *file* begins with a dash (–), the end of the command line options must be marked with ––.

file See **Format 1**.

Format 3, Format 4: List information on the invoking user

who am i

who am I

Identical to who –m. who lists:

- the login name under which you logged in
- the device name (without the /dev/) of the terminal on which you logged in
- the login time

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

The LC_TIME environment variable governs the format of date and time strings.

If LC_MESSAGES or LC_TIME is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

Listing general information on all users currently logged in by calling who without options.

```
$ who
martin      tty001       Mar  7 10:53
moira       tty002       Mar  7 11:10
```

Example 2

Listing detailed information with column headings on all users currently logged in by calling who with the –H and –u options.

```
$ who -Hu
NAME        LINE         TIME           IDLE     PID  COMMENTS
```

```
martin     tty001       Mar  7 10:53   .      6252
moira      tty002       Mar  7 11:10  0:22   6289
```

Example 3

Listing concise information on all users currently logged in by calling `who` with the `-q` option.

```
$ who -q
martin    moira
# users=2
```

Example 4

Listing information on the invoking user with `who am i`.

```
$ who am i
martin   tty001       Mar  7 10:53
```

FILES

`/var/adm/utmp`
: File from which `who` obtains its information by default. This file records the latest status.

`/var/adm/wtmp`
: This file can be used as an alternative to `/var/adm/utmp`. This file is regularly cleared by the system administrator.

`/etc/inittab`
: Configuration file for the `init` system process.

SEE ALSO

`date(1)`, `login(1)`, `mesg(1)`, `su(1)`, `whoami(1)`, `init(1M)`, `wait(2)`, `inittab(4)`, `utmp(4)`.

User's Reference Manual

whoami(1)
BSD Compatibility Package

NAME

`whoami` - display the effective current username

SYNOPSIS

`/usr/ucb/whoami`

DESCRIPTION

`whoami` displays the login name corresponding to the current effective user ID. If you have used `su` to temporarily adopt another user, `whoami` will report the login name associated with that user ID. `whoami` gets its information from the `geteuid` and `getpwuid` library routines (refer to `getuid` and `getpwent`).

FILES

`/etc/passwd`
Username data base

SEE ALSO

`su`(1), `who`(1), `getuid`(2), `getpwent`(3C).

User's Reference Manual

write(1)

NAME

`write -` write to another user

SYNOPSIS

`write` *recipient* [*ttyname*] □
text ...□
... □
<CTRL-D>

DESCRIPTION

`write` is used to send messages to another user. `write` reads one line at a time from standard input and echoes the lines read as messages on the specified user's terminal. When `write` is invoked, a message header initially appears on the recipient's terminal indicating the name of the sender, the terminal, and the time at which the message is sent. This is followed by the actual messages.

Users may communicate interactively by sending messages to each other with `write` (see **Two-way conversations**).

Before the call

Users who have denied other users permission to write to their terminals (with `mesg -n`) cannot be written to with `write`. A user with system administrator privileges can write to any terminal.

OPERANDS

recipient

Login name of a user logged in to a terminal. You can also send yourself messages. If a user is logged in to more than one terminal at the same time, the appropriate terminal can be specified as well.

The command `who` lists all currently logged-in users and their terminals.

ttyname

Number of the terminal on which the recipient is logged in.

ttyname not specified:

`write` searches the file `/var/adm/utmp` for the terminal. If several entries exist because a user is logged in more than once, `write` uses the first entry recorded in the file. The following message is displayed:

`user is logged on more than one place.`□
`You are connected to "`*ttyname*`".`□
`Other locations are:`*ttyname*

text Text to be sent as a message. `write` reads from standard input one line at a time up to an end-of-file signal:

- A line that begins with an exclamation mark (!) is interpreted as a command; `write` passes everything that follows the ! in the line to the shell. The command is then executed, while `write` remains in an active state. Any output that the command writes to standard output will be included in the text to be sent as a message.
- All other normal lines are sent to the recipient as a message.
- When `write` encounters the end-of-transmission character ("EOT\n"), it displays this character on the other terminal and then exits.
- Non-printing characters are converted before they are sent. Control character will appear as a caret ^ followed by the appropriate ASCII character; characters with the eighth bit set will appear in meta notation. For example, '\003' is displayed as '^C' and '\372' as 'M-z'.

Two-way conversations

Users may communicate interactively by sending messages to each other with `write`. Conversation between two users proceeds as follows:

1. User A invokes `write` with the login name of user B. User B receives an announcement (message header) informing him or her that user A wishes to communicate something.

   `Message from` *sender* `(`*terminal*`) [`*time*`]`

   User A hears two bell tones to indicate that the connection has been set up and that user B is able to receive.
2. User B therefore calls `write` with the login name of user A.

   `write` *sender* `[`*terminal*`]`

   The message header received by user A serves as an acknowledgement.
3. Both users can now communicate with one another. Each user should end a message with a distinctive signal so that the other knows when to reply. There should also be a distinctive signal to indicate when the conversation is to be terminated.
4. The conversation can be terminated by pressing the <CTRL-D> or the <DEL> key. The command `mesg -n` can then be used if other users are to be denied permission to write further messages.

ERROR MESSAGES

`user is not logged on` or `user is not at "`*tty*`"`

The addressed user has not logged in.

`Permission denied.`

The addressed user has denied other users permission to write to his/her terminal [see `mesg(1)`].

`Warning: You have your terminal set to "mesg –n". No reply possible.`

Your terminal cannot receive messages from other users.

`Can no longer write to ttyname`

The user of the other terminal has withdrawn write permission since transmission started [see `mesg(1)`].

LOCALE

The `LC_MESSAGES` environment variable governs the language in which message texts are displayed. If `LC_MESSAGES` is undefined or is defined as the null string, it defaults to the value of `LANG`. If `LANG` is likewise undefined or null, the system acts as if it were not internationalized.

The `LC_ALL` environment variable governs the entire international environment. `LC_ALL` takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

Send a message to user `karen`:

```
$ write karen
!date
Mon Oct 15 19:00:13 MET 1990
Today, at 15.10,
I decided to quit
smoking! <CTRL-D>
```

FILES

`/var/adm/utmp`
: The file in which all logged in users are registered.

`/usr/bin/sh`
: Command interpreter for lines beginning with `!`.

SEE ALSO

mail(1), mailx(1), mesg(1), pr(1), sh(1), talk(1), who(1).

User's Reference Manual

xargs(1)

NAME

`xargs` - construct argument list(s) and execute command

SYNOPSIS

`xargs` [*option ...*] [*command* [*initial_argument ...*]]

DESCRIPTION

`xargs` combines command line arguments with arguments that it reads from standard input and executes the specified command one or more times. The number of arguments read for each command invocation and the manner in which they are combined can be determined by the options specified.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or newline characters; empty lines are discarded. Blanks and tabs may be embedded as part of an argument only if they are either escaped with a backslash \ or if the argument is enclosed in single or double quotes ('...' or "..."). Otherwise, they are interpreted as separators between arguments. The usual quoting mechanisms apply as well, i.e. metacharacters are also escaped as mentioned above.

OPTIONS

Options `–i` or `–I`, `–l` or `–L` and `–n` determine how the command line arguments and the arguments read from standard input are to be selected for each invocation of *command*.

If none of the `–I` (or `–i`), `–L` (or `–l`), or `–n` options is used:

The initial arguments specified in the `xargs` command line are followed by arguments read continuously from standard input until an internal buffer is full. *command* is then executed with the accumulated arguments. This process is repeated until there are no more arguments.

If combinations of the `–I` (or `–i`), `–l` (or `–l`), or `–n` options are used:

When there are conflicts (e.g. `–l` and `–n`), the last specified option takes effect.

`–I` *repstr*

Insert mode; *command* is executed for each line read from standard input, taking each line as an argument and inserting it for each occurrence of *repstr* in the list of initial arguments specified when `xargs` was called.

A maximum of 5 arguments in the list of initial arguments may each contain one or more instances of *repstr*. Blanks and tabs at the beginning of each line are ignored. Constructed arguments may not grow larger than 255 characters.

The `–I` option automatically also sets the `–x` option.

This corresponds to the old option `–i`, which is still supported. For *repstr*, braces { } are assumed in the case of `–i` if not explicitly specified.

`–L` *number*

command is executed for every *number* non-empty argument lines that `xargs` reads from standard input. If fewer than *number* lines remain when *command* is invoked for the last time, *command* will be executed with these lines.

A line is considered to end with the first newline character in it unless the last character in the line is a blank or a tab. A trailing blank or tab signals continuation to the next non-empty line.

This corresponds to the old option `–l`, which is still supported. If *number* is not specified for `–l`, the value 1 is assumed. `–x` is set automatically for `–l`.

`–n` *number*

Executes *command* using as many standard input arguments as possible (up to a maximum of *number*). Fewer arguments will be used if their total size is greater than *maxsize* characters (see `–s` option). If fewer than *number* arguments remain for the last invocation, these arguments will be used. If option `–x`

is also set, each *number* arguments must not exceed the maximum limitation specified with *maxsize* (see option –s), as otherwise xargs will terminate execution.

–E *eofstr*

Defines *eofstr* as the logical end-of-file string. xargs reads from standard input until either the actual end-of-file or the logical *eofstr* is encountered.

Specify a string for *eofstr*. This string is interpreted as the logical end-of-file string when xargs is executed.

This corresponds to the old option –e, which is still supported. If –e is specified without *eofstr*, a logical end-of-file string is no longer specified. The underscore _ has no special meaning and is processed as a normal character.

Neither –E nor –e specified: The underscore _ is interpreted as the logical end-of-file.

–p Prompt mode; the user is asked whether to execute *command* at each invocation. In addition, the trace mode (option –t) is turned on to print the invoked *command*, followed by a ?... prompt.

–s *maxsize*

Sets the maximum number of characters in each argument list to *maxsize*. An argument list is a combination of arguments formed on the basis of rules defined by the –I (or –i), –L (or –l), or –n options. Note that the character count for *maxsize* includes one extra character for each argument and the count of characters in the command name.

–t Trace mode; the *command* and each constructed argument list are echoed to standard error just prior to their execution.

–x Causes xargs to terminate if any argument list would exceed the specified maximum of *maxsize* characters (see option –s).

The –x option is forced when options –I (or –i) and –l are set.

When none of the –I (or –i), –L (or –l), or –n options is coded, the execution of xargs is terminated if the total length of all arguments exceeds *maxsize* characters (see option s).

–– If *command* begins with a dash (–), the end of the command line options must be marked with ––.

command

Any command may be specified for *command*. xargs will terminate if it receives an exit status of 255 from *command* or cannot execute it. If *command* is a shell script, it should include an *exit* command which explicitly returns a suitable exit status so as to avoid accidentally returning with 255 (e.g. exit –1).

command not specified: The command *echo* is assumed for *command*.

initial_argument

Argument lists are constructed (as described above) by combining initial arguments specified on the xargs command line with arguments read from the standard input (see options –I, –L, and –n). The specified *command* is then executed with these argument lists.

The initial arguments always appear at the beginning of an argument list, except when the –I or –i option is set.

initial_argument not specified: Argument lists are only constructed from arguments read from the standard input.

EXIT STATUS

0 Each call of *command* returned the exit status 0.

1-125

The arguments could not be combined as requested, or at least one call of *command* returned a non-zero exit status or a further error occurred.

126 The specified *command* exists, but cannot be executed.

127 The specified *command* cannot be found.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed.

LC_COLLATE governs the collating sequence.

LC_CTYPE governs character classes and character conversion (shifting).

If LC_MESSAGES, LC_COLLATE or LC_CTYPE is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

If any of the locale variables has an invalid value, the system acts as if none of the variables was set.

EXAMPLES

Example 1

The following shell script named relocate moves files with names that do not begin with a period (.) from one directory to another:

```
$ cat relocate
ls $1 | xargs –I {} –t mv $1/{} $2/{}
$ relocate dir1 dir2
mv dir1/file1 dir2/file1
mv dir1/file2 dir2/file2
mv dir1/file3 dir2/file3
```

The two positional parameters $1 and $2 are assigned the values of the arguments specified (dir1 and dir2) when relocate is invoked. The ls $1 command displays the contents of the dir1 directory with one file name in each line. These file names are sequentially inserted for {} (option –I {}). Each mv command is echoed with its argument list (option –t) just before it is invoked.

Example 2

The following shell script named who_and_when collects the output of the parenthesized commands (...) in one line and appends this line to the end of the file log:

```
$ cat who_and_when
(logname; date; echo $0 $*) | xargs >>log
$ cat log
michael Wed Mar 29 14:21:06 MET 1989 who_and_when
```

Example 3

The following shell script named archive places files in the current directory whose names do not begin with a period (.) in an archive named archive.a [see ar(1)]:

```
$ cat archive
ls | xargs -p -L 1 ar r archive.a
$ archive
ar r archive.a file1 ?... y
ar: creating archive.a
ar r archive.a file2 ?... n
ar r archive.a file3 ?... y
ar r archive.a file4 ?... n
$ ar t archive.a
file1
file3
```

ls displays the contents of the current directory on standard output with one file name in each line. xargs then calls ar with the arguments r, archive.a, and each file name returned by ls. Since the –p option has been set, you are asked whether or not the corresponding ar command is to be executed in each case. If you respond with a "y" the first time, ar creates the archive file archive.a, issues a corresponding message, and saves the current file in it. Your subsequent responses to each prompt will then determine which files are to be stored in archive.a. If required, the contents of archive.a can be listed at the end with the ar –t

command.

SEE ALSO

echo(1) .

User's Reference Manual

xpg4_sh(1)

NAME

`xpg4_sh` - XPG4 shell

SYNOPSIS

`/sbin/xpg4/sh` [*option ...*] [*file*] [*argument ...*]

DESCRIPTION

The XPG4 shell is practically identical to the Korn shell [see `ksh(1)`]. However, since the Korn shell does not fulfill all the conditions specified by X/Open (see the *CAE Specification, Commands and Utilities, Issue 4*), a shell is also supplied under `/sbin/xpg4/sh`, which satisfies the remaining XPG4 conditions (X/Open Branding).

The `ksh` Korn shell was not fully adapted to XPG4 because, in many cases, it should behave more like the `sh` Bourne shell. To prevent the behavior of the Korn shell from being incompatible with its previous behavior in such cases, the changes were carried out separately for a special XPG4 shell.

The default shell (`/sbin/sh`) is currently linked to the Bourne shell (`/sbin/bourne/sh`). If you wish to have the XPG4 shell as the default shell, you need to set up the link to `/sbin/xpg4/sh`.

Differences between the Korn shell and the XPG4 shell

1. If you do not enter anything after the `-o` option when calling the shell, the XPG4 shell sends the following message to standard error: `ksh: '': bad option(s).`

   The Korn shell, however, sends the permitted options to standard output.

2. In the Korn shell, `time` is a keyword at the start of a line (like `if`, `then` etc.; cf. `ksh(1)` , section **Reserved words**). In the XPG4 shell (as with the Bourne shell), on the other hand, `time` is a standard user command (`/usr/bin`). The following deviations regarding `time` can be distinguished.

   The behavior of the

   `time` *pipeline*

   command varies. In the case of the Korn shell, the pipeline commands are executed and then the elapsed time, the user and system time for these commands are reported as statistics to standard error.

   With the XPG4 shell, `time` is only executed for the first pipeline command and the statistics are output beforehand.

   Thus in the case of the Korn shell, if you issue the

   ```
   time ls | pg
   ```

   call, you will first see the output of `ls` and the `pg` input must then be performed. It is only when everything has ended that the statistics are output for `ls` and `pg`.

   With the XPG4 shell, the statistics are output first, followed by the output of `ls` and the input for `pg`.

3. With the Korn shell, an error message is output if the `echo` command cannot write when writing to a file because the disk is full. The end status in this case is 1. The XPG4 shell ignores this error.

4. The Korn shell does not lose any parameters in the following `getopts` command if the shell script is called with, for example, `-l abc`. When `echo $*` is called for the second time, the XPG4 shell only outputs `-l` instead of `-l abc`:

   ```
   #!/sbin/ksh
   set -x
   echo $*
   while getopts :l:u:t: arguments
   do
     case $arguments in
   ```

```
        l) ;;
        u) ;;
        t) ;;
        *) echo "wrong option \n"
 exit 1
 ;;
  esac
done
echo $*
```

5. If an attempt is made with a read-only variable (`typeset -r`) to change the value, the Korn and XPG4 shells return the error message " `...: is read only`".

   The difference between the two shell responses is that with the XPG4 shell, the error message is deliberately generated later.

6. The Korn shell supplies the function name for a shell script function in `$0` while the XPG4 shell also supplies the name of the shell script.

   For example, in the following shell script called `test.sh`:

   ```
   typeset -fux dummy
   function dummy
   {
       echo $0
   }

   dummy
   ```

   The Korn shell echoes `dummy`, the XPG4 shell echoes `test.sh`.

SEE ALSO

ksh(1), sh(1).

User's Reference Manual

yes(1)

NAME

`yes` - prints string repeatedly

SYNOPSIS

`yes` [*string*]

DESCRIPTION

`yes` repeatedly outputs "y", or if a single string argument is given, *string* is output repeatedly. The command will continue indefinitely unless aborted.

`yes` is useful in pipes to commands that prompt for input and require a "y" response for a yes. In this case, `yes` terminates when the command that it pipes to terminates, so that no infinite loop occurs.

EXAMPLES

The `checkin-list` file contains a list comprising a number of file names, which are to be checked into SRCS administration using the interactive command `ci`. The easiest way of doing this is to use a shell script in which the `checkin-list` is read and the required input "y" is retrieved using the `yes` command. The script might appear as follows:

```
for i in `cat checkin-list`; do
          echo $i
          yes | ci -Nv543a -m"info-text" $i
done
```

User's Reference Manual

zcat(1)

NAME

zcat - display compressed files in expanded form

SYNOPSIS

zcat [*file* ...]

DESCRIPTION

zcat takes files that have been compressed with compress(1) and copies them to standard output in their original form. The compressed file remains compressed. Files compressed with compress can be restored to their original form with the uncompress(1) command.

zcat, together with compress and uncompress, forms a group of commands used to compress and expand files and to display files in expanded form. zcat is identical to uncompress –c. Similar services are provided by another group of commands, pack(1), unpack(1) and pcat(1).

OPERANDS

file Name of the compressed file to be output in its original form. You can name any number of files.

You may enter the name of the file with or without the .Z extension; if you omit the extension, zcat automatically searches for the appropriate .Z file.

ERROR MESSAGES

The following errors result in the failure of the zcat command.

filename: no such file or directory
: The file you specified does not exist.

filename: not in compressed format
: The file you specified was not compressed using compress.

filename: compressed with *xx*bits, can only handle *yy*bits
: The file was compressed by a program which could handle more bits than the compression code on this machine. You may be able to recompress the file with a smaller number of bits.

uncompress: corrupt input
: There has been a SIGSEGV signal (addressing error due to segmentation violation). This usually means that the input file is corrupted.

LOCALE

The LC_MESSAGES environment variable governs the language in which message texts are displayed. If LC_MESSAGES is undefined or is defined as the null string, it defaults to the value of LANG. If LANG is likewise undefined or null, the system acts as if it were not internationalized.

The LC_ALL environment variable governs the entire locale. LC_ALL takes precedence over all the other environment variables that affect internationalization.

EXAMPLES

The file zcat_ex is printed with cat, compressed with compress, and then output again with zcat.

```
$ zcat zcat_ex
I have still not been compressed!
$ compress -fv zcat_ex
zcat_ex: Compression: -16.12% -- replaced with zcat_ex.Z
$ zcat zcat_ex.Z
(Contents of file)
```

SEE ALSO

cat(1), compress(1), pack(1), pcat(1), uncompress(1), unpack(1).