68NW9209H41A

SYSTEM V/88 Release 3.2
User's Reference Manual

SYSTEM V/88 Release 3.2

User's

Reference Manual

(68NW9209H41A)

PREFACE

The *3User's Reference Manual* describes the commands that constitute the basic software for the SYSTEM V/88 Release 3.2 software.

First Edition February 1990

CONTENTS

1. Commands

NAME

intro – introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, commands available with the SYSTEM V/88 operating system. Certain distinctions of purpose are made in the headings.

The following Utility packages are part of the system:

Basic Networking Utilities (BNU)
Directory and File Management Utilities
Editing Utilities
Essential Utilities
Help Utilities
Inter-process Communications (IPC)
Line Printer Spooling Utilities
Performance Measurement Utilities
Security Administration Utilities
Spell Utilities
User Environment Utilities
Networking Support Utilities
Remote File Sharing Utilities (RFS)

Manual Page Command Syntax

Unless otherwise noted, commands described in the SYNOPSIS section of a manual page accept options and other arguments according to the following syntax and should be interpreted as explained below.

name [–*option*...] [*cmdarg*...]

where:

[ ]	Surround an *option* or *cmdarg* that is not required.
...	Indicates multiple occurrences of the *option* or *cmdarg*.
name	The name of an executable file.
option	(Always preceded by a "–".) *noargletter*... or, *argletter optarg*[,...]
noargletter	A single letter representing an option without an option-argument. Note that more than one *noargletter* option can be grouped after one "–" (see Rule 5).

argletter A single letter representing an option requiring an option-argument.

optarg An option-argument (character string) satisfying a preceding *argletter*. Note that groups of *optargs* following an *argletter* must be separated by commas, or separated by white space and quoted (see Rule 8).

cmdarg Pathname (or other command argument) *not* beginning with "–", or "–" by itself indicating the standard input.

Command Syntax Standard: Rules

These command syntax rules are not followed by all current commands, but all new commands will obey them. *getopts* (1) should be used by all shell procedures to parse positional parameters and to check for legal options; it supports Rules 3-10 below. The enforcement of the other rules must be done by the command itself.

1. Command names (*name* above) must be between two and nine characters long.
2. Command names must include only lowercase letters and digits.
3. Option names (*option* above) must be one character long.
4. All options must be preceded by "–".
5. Options with no arguments may be grouped after a single "–".
6. The first option-argument (*optarg* above) following an option must be preceded by white space.
7. Option-arguments cannot be optional.
8. Groups of option-arguments following an option must either be separated by commas or separated by white space and quoted (e.g., –o xxx,z,yy or –o "xxx z yy").
9. All options must precede operands (*cmdarg* above) on the command line.
10. "—" may be used to indicate the end of the options.
11. The order of the options relative to one another should not matter.
12. The relative order of the operands (*cmdarg* above) may affect their significance in ways determined by the command with which they appear.

13. "–" preceded and followed by white space should only be used to mean standard input.

SEE ALSO

getopts(1).
exit(2), wait(2), getopt(3C) in the *Programmer's Reference Manual.*

DIAGNOSTICS

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

WARNINGS

Some commands produce unexpected results when processing files containing NULL characters. These commands often treat text input lines as strings and therefore become confused upon encountering a NULL character (the string terminator) within a line.

NAME

300, 300s – handle special functions of DASI 300 and 300s terminals

SYNOPSIS

300 [**+12**] [**–n**] [**–d***t*,*l*,*c*]

300s [**+12**] [**–n**] [**–d***t*,*l*,*c*]

DESCRIPTION

The *300* command supports special functions and optimizes the use of the DASI 300 (GSI 300 or DTC 300) terminal; *300s* performs the same functions for the DASI 300s (GSI 300s or DTC 300s) terminal. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. In the following discussion of the *300* command, it should be noted that unless your system contains the DOCUMENTER'S WORKBENCH Software, references to certain commands (e.g., *nroff*, *neqn*, *eqn*) will not work. It also attempts to draw Greek letters and other special symbols. It permits convenient use of 12 pitch text and reduces printing time 5 to 70%. The *300* command can be used to print equations neatly, in the sequence:

neqn file ... | nroff | 300

CAUTION. Make sure the **PLOT** switch on your terminal is ON before *300* is used.

The behavior of *300* can be modified by the optional flag arguments to handle 12 pitch text, fractional line spacings, messages, and delays:

+12
permits use of 12 pitch, 6 Lines Per Inch (LPI) text. DASI 300 terminals normally allow only two combinations: 10 pitch, 6 LPI, or 12 pitch, 8 LPI. To obtain the 12 pitch, 6 LPI combination, you should turn the **PITCH** switch to 12, and use the **+12** option.

–n
controls the size of half-line spacing. A half-line is, by default, equal to 4 vertical plot increments. Because each increment equals 1/48 of an inch, a 10 pitch linefeed requires 8 increments, while a 12 pitch linefeed needs only 6. The first digit of *n* overrides the default value, which allows for individual taste in the appearance of subscripts and superscripts. For example, *nroff* half-lines could be made to act as quarter-lines by using **–2**. You can also obtain appropriate half-lines for 12 pitch, 8 LPI mode by using the option **–3** alone by setting the **PITCH** switch to 12 pitch.

–d*t*,*l*,*c*

controls delay factors. The default setting is **–d3,90,30**. DASI 300 terminals sometimes produce peculiar output when faced with very long lines, too many tab characters, or long strings of blankless, non-identical characters. One **NULL** (delay) character is inserted in a line for every set of *t* tabs, and for every contiguous string of *c* non-blank, non-tab characters. If a line is longer than *l* bytes, 1+(total length)/20 **NULL**s are inserted at the end of that line. Items can be omitted from the end of the list, implying use of the default values. Also, a value of zero for *t* (*c*) results in two **NULL** bytes per tab (character). The former may be needed for C programs, the latter for files like **/etc/passwd**. Because terminal behavior varies according to the specific characters printed and the load on a system, the user may have to experiment with these values to get correct output. The **–d** option exists only as a last resort for those few cases that do not otherwise print properly. For example, the file **/etc/passwd** may be printed using **–d3,30,5**. The value **–d0,1** is a good one to use for C programs that have many levels of indentation.

Note that the delay control interacts heavily with the prevailing carriage return and linefeed delays. The *stty*(1) modes **nl0 cr2** or **nl0 cr3** are recommended for most uses.

The *300* command can be used with the *nroff* –s flag or **.rd** requests, when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the **RETURN** key in these cases, you must use the **LINEFEED** key to get any response.

In many (but not all) cases, the following sequences are equivalent:

nroff –T300 files ... and nroff files ... | 300
nroff –T300–12 files ... and nroff files ... | 300 +12

The use of *300* can thus often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of *300* may produce better-aligned output.

SEE ALSO

450(1), mesg(1), stty(1), tabs(1).

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.

If your output contains Greek and/or reverse linefeeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters and misaligning the first line of text after one or more reverse linefeeds.

NAME

4014 – paginator for the Tektronix 4014 terminal

SYNOPSIS

4014 [**–t**] [**–n**] [–c*N*] [–p*L*] [*file*]

DESCRIPTION

The output of *4014* is intended for a Tektronix 4014 terminal; *4014* arranges for 66 lines to fit on the screen, divides the screen into *N* columns, and contributes an eight-space page offset in the (default) single column case. Tabs, spaces, and backspaces are collected and plotted when necessary. Teletype Model 37 half- and reverse-line sequences are interpreted and plotted. At the end of each page, *4014* waits for a new-line (empty line) from the keyboard before continuing on to the next page. In this wait state, the command !*cmd* will send the *cmd* to the shell.

The command line options are:

–t
Do not wait between pages (useful for directing output into a file).

–n
Start printing at the current cursor position and never erase the screen.

–c*N*
Divide the screen into *N* columns and wait after the last column.

-p*L* [*i* or *l*]
Set page length to *L*. *L* accepts the scale factors *i* (inches) and *l* (lines). Default is lines.

It should be noted that unless your system contains the DOCUMENTER'S WORKBENCH Software, references to certain commands (e.g., *troff*, *neqn*, *eqn*) do not pertain.

SEE ALSO

pr(1).

NAME

450 – handle special functions of the DASI 450 terminal

SYNOPSIS

450

DESCRIPTION

The *450* command supports special functions of, and optimizes the use of, the DASI 450 terminal, or any terminal that is functionally identical, such as the Diablo 1620 or Xerox 1700. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. It also attempts to draw Greek letters and other special symbols in the same manner as *300*(1). It should be noted that, unless your system contains DOCUMENTER'S WORKBENCH Software, certain commands (e.g., *eqn*, *nroff*, *tbl*) will not work. Use *450* to print equations neatly, in the sequence:

neqn file ... | nroff | 450

CAUTION. Make sure the PLOT switch on your terminal is ON before *450* is used. The SPACING switch should be put in the desired position (either 10 or 12 pitch). In either case, vertical spacing is 6 LPI, unless dynamically changed to 8 LPI by an appropriate escape sequence.

Use *450* with the *nroff* **–s** flag or **.rd** requests when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the RETURN key in these cases, you must use the LINEFEED key to get any response.

In many (but not all) cases, the use of *450* can be eliminated in favor of one of the following:

nroff –T450 files ...

or

nroff –T450–12 files ...

The use of *450* can thus often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of *450* may produce better-aligned output.

The *neqn* names of, and resulting output for, the Greek and special characters supported by *450* are shown in *greek*(5).

SEE ALSO

300(1), mesg(1), stty(1), tabs(1).

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.

If your output contains Greek and/or reverse `LINEFEEDS`, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters and misaligning the first line of text after one or more reverse `LINEFEEDS`.

NAME

acctcom – search and print process accounting file(s)

SYNOPSIS

acctcom [[*options*][*file*]] . . .

DESCRIPTION

acctcom reads *file*, the standard input, or */usr/adm/pacct*, in the form described by *acct*(4) and writes selected records to the standard output. Each record represents the execution of one process. The output shows the **COMMAND NAME, USER, TTYNAME, START TIME, END TIME, REAL (SEC), CPU (SEC), MEAN SIZE(K),** and optionally, *F* (*fork*/*exec* flag: **1** for *fork* without *exec*), **STAT** (system exit status), **HOG FACTOR, KCORE MIN, CPU FACTOR, CHARS TRNSFD,** and **BLOCKS READ** (total blocks read and written).

A **#** is prepended to the command name if the command was executed with superuser privileges. If a process is not associated with a known terminal, a **?** is printed in the **TTYNAME** field.

If no *files* are specified, and if the standard input is associated with a terminal or **/dev/null** (as is the case when using **&** in the shell), **/usr/adm/pacct** is read; otherwise, the standard input is read.

If any *file* arguments are given, they are read in their respective order. Each file is normally read forward, i.e., in chronological order by process completion time. The file **/usr/adm/pacct** is usually the current file to be examined; a busy system may need several such files of which all but the current file are found in **/usr/adm/pacct**. The *options* are:

–a
Show some average statistics about the processes selected. The statistics will be printed after the output records.

–b
Read backwards, showing latest commands first. This *option* has no effect when the standard input is read.

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

–h
Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution. This "hog factor" is computed as:

(total CPU time)/(elapsed time).

–i
Print columns containing the I/O counts in the output.

–k
Instead of memory size, show total kcore-minutes.

–m
Show mean core size (the default).

–r
Show CPU factor (user time/(system-time + user-time).

–t
Show separate system and user CPU times.

–v
Exclude column headings from the output.

–l *line*
Show only processes belonging to terminal **/dev/line**.

–u *user*
Show only processes belonging to *user* that may be specified by: a user ID, a login name that is then converted to a user ID, a **#** (which designates only those processes executed with superuser privileges), or a **?** (which designates only those processes associated with unknown user IDs). Remember that the **#** or the **?** character is enclosed within apostrophes or quotation marks or preceded by a backslash.

–g *group*
Show only processes belonging to *group*. The *group* may be designated by either the group ID or group name.

–s *time*
Select processes ending at or after *time*, given in the format *hr* [:*min* [:*sec*]].

–e *time*
Select processes starting 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 matching *pattern* that may be a regular expression as in *ed*(1) except that + means one or more occurrences.

–q
Do not print any output records, just print the average statistics as with the **–a** option.

–o *ofile*
Copy selected process records in the input data format to *ofile*; suppress standard output printing.

–H *factor*
Show only processes that exceed *factor*, where factor is the "hog factor" as 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 plus user, exceeding *sec* seconds.

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

FILES

/etc/passwd
/usr/adm/pacct
/etc/group

SEE ALSO

ps(1), su(1)
acct(2), acct(4), utmp(4) in the *Programmer's Reference Manual*.
acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), fwtmp(1M), runacct(1M) in the *System Administrator's Reference Manual*.

BUGS

acctcom reports only on processes that have terminated; use *ps*(1) for active processes. If *time* exceeds the present time, *time* is interpreted as occurring on the previous day.

NAME

admin – create and administer SCCS files

SYNOPSIS

admin [**–n**] [**–i**[*name*]] [**–r***rel*] [**–t**[*name*]] [**–f***flag*[*flag-val*]] [**–d***flag*[*flag-val*]] [**–a***login*] [**–e***login*] [**–m**[*mrlist*]] [**–y**[*comment*]] [**–h**] [**–z**] *files*

DESCRIPTION

admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with –, and named files (note that SCCS file names must begin with the characters **s.**). If a named file does not exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The following describes the keyletter arguments. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

–n

This keyletter indicates to create a new SCCS file.

–i*[name]*

The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see **–r** keyletter for delta numbering scheme). If the **i** keyletter is used but the file name is omitted, the text is obtained by reading the standard input until an EOF is encountered. If this keyletter is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an *admin* command on which the **i** keyletter is supplied. Using a single *admin* to create two or more SCCS files requires that they be created empty (no **–i** keyletter). Note that the **–i** keyletter implies the **–n** keyletter.

–r*rel*

The *rel*ease into which the initial delta is inserted. This keyletter may be used only if the **–i** keyletter is also used. If the **–r** keyletter is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

–t*[name]*

The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **–t** keyletter is used and *admin* is creating a new SCCS file (the **–n** and/or **–i** keyletters also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **–t** keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **–t** keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

–f*flag*

This keyletter specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **f** keyletters may be supplied on a single *admin* command line. The allowable *flag*s and their values are:

b

Allows use of the **–b** keyletter on a *get*(1) command to create branch deltas.

c*ceil*

The highest release (i.e., "ceiling"), a number greater than 0 but less than or equal to 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **c** flag is 9999.

f*floor*

The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(1) command for editing. The default value for an unspecified **f** flag is 1.

d*SID*

The default delta number (SIDs+1) to be used by a *get*(1) command.

i*[str]*

Causes the **No id keywords (ge6)** message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(1)) are found in the text retrieved or stored in the SCCS file. If a value is supplied, the keywords must exactly match the given string, however the string must contain a keyword, and no embedded newlines.

j

Allows concurrent *get*(1) commands for editing on the same SIDs+1 of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list*

A *list* of releases to which deltas can no longer be made (**get –e** against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= | **a**

The character **a** in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n

Causes *delta*(1) to create a **NULL** delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These **NULL** deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text*

User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*(1).

m*mod*

*mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*(1). If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type*
type of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*(1).

v*pgm*
Causes *delta*(1) to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program (see *delta*(1)). (If this flag is set when creating an SCCS file, the **m** keyletter must also be used even if its value is `NULL`).

–d*flag*
Causes removal (deletion) of the specified *flag* from an SCCS file. The **–d** keyletter may be specified only when processing existing SCCS files. Several **–d** keyletters may be supplied on a single *admin* command. See the **–f** keyletter for allowable *flag* names.

l*list*
A *list* of releases to be "unlocked". See the **–f** keyletter for a description of the **l** flag and the syntax of a *list*.

–a*login*
A *login* name, or numerical group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several **a** keyletters may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a **!**, they are to be denied permission to make deltas.

–e*login*
A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several **e** keyletters may be used on a single *admin* command line.

–m*[mrlist]*
The list of (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(1). The **v** flag must be set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). Diagnostics will occur if the **v** flag is not set or MR validation fails.

–y*[comment]*

The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(1). Omission of the **–y** keyletter results in a default comment line being inserted in the form:

date and time created *YY/MM/DD HH:MM:SS* by *login*

The **–y** keyletter is valid only if the **–i** and/or **–n** keyletters are specified (i.e., a new SCCS file is being created).

–h

Causes *admin* to check the structure of the SCCS file (see *sccsfile*(5)), and to compare a newly computed checksum (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced. keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

–z

The SCCS file checksum is recomputed and stored in the first line of the SCCS file (see **–h**, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

The last component of all SCCS file names must be of the form s.*file-name*. New SCCS files are given mode 444 (see *chmod*(1)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called x.*file-name*, (see *get*(1)), created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of *ed*(1). *Care must be taken!* The edited file should *always* be processed by an **admin –h** to check for corruption followed by an **admin –z** to generate a proper check-sum. Another **admin –h** is recommended to ensure the SCCS file is valid.

admin also makes use of a transient lock file (called **z.***file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

delta(1), get(1), prs(1), what(1), sccsfile(4).
ed(1), help(1) in the *User's Reference Manual*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ar – archive and library maintainer for portable archives

SYNOPSIS

ar key [*posname*] *afile* [*name*] ...

DESCRIPTION

The *ar* command maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by *ar* consist of printable ASCII characters. If an archive is 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. The portable archive format and structure is described in detail in *ar*(4). The archive symbol table (see *ar*(4)) is used by the link editor (*ld*(1)) to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by *ar* when there is at least one object file in the archive. The archive symbol table is in a specially named file which is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever the *ar*(1) command is used to create or update the contents of such an archive, the symbol table is rebuilt. The **s** option described below will force the symbol table to be rebuilt.

Unlike command options, the command key is a required part of *ar*'s command line. The key (which may begin with a –) is formed with one of the following letters: **drqtpmx**. Arguments to the *key*, alternatively, are made with one of more of the following set: **vuaibcls**. *posname* is an archive member name used as a reference point in positioning other files in the archive. *afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d

Delete the named files from the archive file.

r

Replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with dates of modification later than the archive files are replaced. If an optional positioning character from the set **abi** is used, the *posname* argument must be present and specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise, new files are placed at the end.

q

Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check if the added members are already in the archive. This option is useful to avoid quadratic behavior when creating a large archive piece-by-piece. Unchecked, the file may grow exponentially up to the second degree.

t

Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p

Print the named files in the archive.

m

Move the named files to the end of the archive. If a positioning character is present, the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.

x

Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

The meanings of the key arguments are:

v

Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, give a long listing of all information about the files. When used with **x**, precede each file with a name.

c

Suppress the message that is produced by default when *afile* is created.

l

Place temporary files in the local (current working) directory rather than in the default temporary directory, **TMPDIR.**

s

Force the regeneration of the archive symbol table even if *ar*(1) is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after the *strip*(1) command has been used on the archive.

FILES

$TMPDIR/* temporary files

$TMPDIR is usually **/usr/tmp** but can be redefined by setting the environment variable TMPDIR (see *tempnam*() in *tmpnam*(3S)).

SEE ALSO

ld(1), lorder(1), strip(1), tmpnam(3S), a.out(4), ar(4) in the *Programmer's Reference Manual*.

NOTES

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

/bin/as – assembler driver script

SYNOPSIS

/bin/as [*options*] *filename*

DESCRIPTION

The */bin/as* command both silicon filters and assembles the named file. This driver script allows the silicon filtering to be performed in a transparent fashion which avoids having to make special changes to makefiles or scripts to accommodate the silicon filter pass. The following flags may be specified in any order:

–o *objfile*
Put the output of the assembly in *objfile*. By default, the output file name is formed by removing the **.s** suffix, if there is one, from the input file name and appending a **.o** suffix.

–F*options*
Pass these options on to the silicon-filter. This option is used to specify the *sifilter* command options that will be used by the silicon filter when it is automatically run by */bin/as*. See *sifilter*(1) for a complete list of filter options.

/bin/as is implemented as a script that calls both the silicon filter and the assembler in two passes.

FILES

/usr/tmp/as[A-Z]XXXXX	temporary files
/lib/as	assembler
/bin/sifilter	silicon filter

SEE ALSO

sifilter(1).

NAME

/lib/as – assembler

SYNOPSIS

/lib/as [*options*] *filename*

DESCRIPTION

The */lib/as* command assembles the named file. The following flags may be specified in any order:

–o *objfile*
Put the output of the assembly in *objfile*. By default, the output file name is formed by removing the **.s** suffix, if there is one, from the input file name and appending a **.o** suffix.

–m
Run the **m4** macro processor on the input to the assembler.

–R
Remove (unlink) the input file after assembly is completed.

–V
Write the version number of the assembler being run on the standard error output.

–Y *[md],dir*
Find the **m4** preprocessor (**m**) and/or the file of predefined macros (**d**) in directory *dir* instead of in the customary place.

FILES

/usr/tmp/as[A-Z]XXXXX temporary files

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [*file*...]

DESCRIPTION

asa interprets the output of FORTRAN programs that use ASA carriage control characters. It processes either the *file*s whose names are given as arguments or the standard input if no file names are supplied. The first character of each line is assumed to be a control character; their meanings are:

(blank)	single newline before printing
0	double newline before printing
1	new page before printing
+	overprint previous line.

Lines beginning with other than the above characters are treated as if they began with a space. The first character of a line is *not* printed. If any such lines appear, an appropriate diagnostic appears on standard error. This program forces the first line of each input file to start on a new page.

EXAMPLE

To correctly view the output of FORTRAN programs that use ASA carriage control characters, *asa* could be used as a filter:

a.out | asa | lp

The output, properly formatted and paginated, would be directed to the line printer. FORTRAN output sent to a file could be viewed by:

asa **file**

FILES

/bin/asa

NAME

assist – assistance using SYSTEM V/88 commands

SYNOPSIS

assist [*name*]
assist [**–s**]
assist [**–c** *name*]

DESCRIPTION

The *assist* command invokes the ASSIST menu interface software. The ASSIST menus categorize user commands according to function in a hierarchy. The menus lead to full-screen forms (called command forms) that aid in the execution of a syntactically correct command line. The menus also lead to interactive simulations of commands or concepts (called walkthrus).

If you type *assist* without options, you enter at the top of the menu interface hierarchy. New users may wish to use the **–s** option to select an introductory tutorial explaining how to use the ASSIST software. Options are:

name
invoke an ASSIST-supported command form or walkthru for *name*

–c *name*
invoke the version of *name* that is in your current directory

–s
reinvoke the ASSIST setup module and check or modify your terminal variable; or access the introductory information about ASSIST

When you invoke *assist*, you perform operations within the program by using *assist* commands. To see a list of the *assist* commands, type ^A (**CTRL–a**) or **f8** (**function-key** 8) when you are in *assist*. When you do this, a list of the commands is printed on the terminal screen. The entire set of commands is described in the *Glossary of ASSIST Commands* in the *ASSIST Software User's Guide.*

EXAMPLE

This example illustrates how to invoke a particular command form directly. In this case, *mkdir* is the desired command form.

assist mkdir

FILES

$HOME/.assistrc	information needed by *assist* (e.g., about the terminal you are using)
/usr/lib/assist	default directory containing assist command forms, walkthrus, and executable programs *assist* and *astgen*

NOTES

The first time you invoke *assist* it ignores any options you give and asks for information about the terminal you are using. Once it has saved this information in a file named *.assistrc* in *assist*, it shows you a list of basic *assist* commands and offers you an introduction to ASSIST.

SEE ALSO

astgen(1).
ASSIST Software User's Guide.

NAME

astgen – program for generating/modifying ASSIST menus or command forms

SYNOPSIS

astgen *name*[*.fs*]

DESCRIPTION

astgen is an interactive program to generate information files (ASCII text data files) that define a menu or command form used by the *assist*(1) program.

Both the *astgen* and *assist*(1) programs recognize and process information files whose names are suffixed with three characters: *.fs* . If no *.fs* file exists for the specified name, *astgen* assumes that a new menu or command form is to be created. If *name* is given without *.fs*, *astgen* automatically will create the file: *name .fs*.

Details of how to use *astgen* are given in the *ASSIST Software Development Tools Guide*.

SEE ALSO

assist(1).
ASSIST Software Development Tools Guide.
ASSIST Software User's Guide.

NAME

at, batch – execute commands at a later time

SYNOPSIS

at *time* [*date*] [+ *increment*]
at *–r job*
at *–l* [*job*]

batch

DESCRIPTION

at and *batch* read commands from standard input to be executed at a later time. *at* allows you to specify when the commands should be executed, while jobs queued with *batch* will execute when system load level permits. *at* may be used with the following options:

–r
Removes jobs previously scheduled with *at*.

–l
Reports all jobs scheduled for the invoking user.

Standard output and standard error output are mailed to the user unless they are redirected elsewhere. The shell environment variables, current directory, umask, and ulimit are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use *at* if their name appears in the file **/usr/lib/cron/at.allow**. If the file does not exist, the file **/usr/lib/cron/at.deny** is checked to determine if the user should be denied access to *at*. If neither file exists, only **root** is allowed to submit a job. If **at.deny** is empty, global usage is permitted. The allow/deny files consist of one user name per line; these files can only be modified by the superuser.

The *time* may be specified as 1, 2, or 4 digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning *hour:minute*. A suffix **am** or **pm** may be appended; otherwise a 24-hour clock time is understood. The suffix **zulu** may be used to indicate GMT. The special names **noon, midnight, now**, and **next** are also recognized.

An optional *date* may be specified as either a month name followed by a day number (and possibly year number preceded by an optional comma) or a day of the week (fully spelled or abbreviated to three characters). Two special "days", **today** and **tomorrow** are recognized. If no *date* is given, **today** is assumed if the given hour is greater than the current hour, and **tomorrow** is assumed if it is less. If the given month is less than the current month (and no year is given), next year is assumed.

The optional *increment* is simply a number suffixed by one of the following: **minutes, hours, days, weeks, months,** or **years.** (The singular form is also accepted.)

Thus legitimate commands include:

```
at 0815am Jan 24
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday
```

at and *batch* write the job number and schedule time to standard error.

batch submits a batch job. It is almost equivalent to "at now", but not quite. For one, it goes into a different queue. For another, "at now" will respond with the error message **too late.**

at **-r** removes jobs previously scheduled by *at* or *batch*. The job number is the number given to you previously by the *at* or *batch* command. You can also get job numbers by typing *at* **-l.** You can only remove your own jobs unless you are the superuser.

EXAMPLES

The *at* and *batch* commands read from standard input the commands to be executed at a later time. *sh*(1) provides different ways of specifying standard input. Within your commands, it may be useful to redirect standard output.

This sequence can be used at a terminal:

```
batch
sort filename >outfile
<CTRL-D> (hold down 'control' and depress 'D')
```

This sequence, which demonstrates redirecting standard error to a pipe, is useful in a shell procedure (the sequence of output redirection specifications is significant):

```
batch <<!
sort filename 2>&1 >outfile | mail loginid
!
```

To have a job reschedule itself, invoke *at* from within the shell procedure, by including code similar to the following within the shell file:

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users
/usr/lib/cron/queue	scheduling information
/usr/spool/cron/atjobs	spool area

SEE ALSO

kill(1), mail(1), nice(1), ps(1), sh(1), sort(1).
cron(1M) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Complains about various syntax errors and times out of range.

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [**–F** *re*] [*parameter...*] ['*prog*'] [**–f** *progfile*] [*file...*]

DESCRIPTION

This is a new version of *awk* that provides capabilities unavailable in previous versions. This version will become the default version of *awk* in the next major UNIX system release.

The **–F** *re* option defines the input field separator to be the regular expression *re*.

parameters, in the form x=... y=... may be passed to *awk,* where x and y are *awk* built-in variables (see list below).

awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. The *prog* string must be enclosed in single quotes (') to protect it from the shell. For each pattern in *prog* there may be an associated action performed when a line of a *file* matches the pattern. The set of pattern-action statements may appear literally as *prog* or in a file specified with the **–f** *progfile* option.

Input files are read in order; if there are no files, the standard input is read. The file name – means the standard input. Each input line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is normally made up of fields separated by white space. (This default can be changed by using the FS built-in variable or the **–F** *re* option.) The fields are denoted **$1**, **$2**, ...; **$0** refers to the entire line.

A pattern-action statement has the form:

```
pattern { action }
```

Either pattern or action may be omitted. If there is no action with a pattern, the matching line is printed. If there is no pattern with an action, the action is performed on every input line.

Patterns are arbitrary Boolean combinations (**!**, **||**, **&&**, and parentheses) of relational expressions and regular expressions. A relational expression is one of the following:

expression relop expression
expression matchop regular expression

where a **relop** is any of the six relational operators in C, and a **matchop** is either ~ (contains) or !~ (does not contain). A conditional is an arithmetic expression, a relational expression, the special expression

var **in** array,

or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line has been read and after the last input line has been read respectively.

Regular expressions are as in *egrep* (see *grep*(1)). In patterns, they must be surrounded by slashes. Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions. A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second pattern.

A regular expression may be used to separate fields by using the **–F** *re* option or by assigning the expression to the built-in variable FS. The default is to ignore leading blanks and to separate fields by blanks and/or tab characters. However, if FS is assigned a value, leading blanks are no longer ignored.

Other built-in variables include:

ARGC	command line argument count
ARGV	command line argument array
FILENAME	name of the current input file
FNR	ordinal number of the current record in the current file
FS	input field separator regular expression (default blank)
NF	number of fields in the current record
NR	ordinal number of the current record
OFMT	output format for numbers (default **%.6g**)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)

An action is a sequence of statements. A statement may be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
do statement while ( conditional )
for ( expression ; conditional ; expression ) statement
for ( var in array ) statement
delete array[subscript]
break
continue
{ [ statement ] ... }
expression      # commonly variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next            # skip remaining patterns on this input line
exit [expr]     # skip the rest of the input; exit status is expr
return [expr]
```

Statements are terminated by semicolons, newlines, or right braces. An empty expression-list stands for the whole input line. Expressions take on string or numeric values as appropriate, and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The C operators ++, —, +=, –=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]), or fields. Variables are initialized to the NULL string or zero. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The **print** statement prints its arguments on the standard output, or on a file if *>expression* is present, or on a pipe if | *cmd* is present. The arguments are separated by the current output field separator and terminated by the output record separator. The **printf** statement formats its expression list according to the format (see *printf*(3S) in the *Programmer's Reference Manual*).

awk has a variety of built-in functions: arithmetic, string, input/output, and general.

The arithmetic functions are: *atan2, cos, exp, int, log, rand, sin, sqrt,* and *srand. int* truncates its argument to an integer. *rand* returns a random number between 0 and 1. *srand* (*expr*) sets the seed value for *rand* to *expr* or uses the time of day if *expr* is omitted.

The string functions are:

gsub(for, repl, in)
behaves like *sub* (see below), except that it replaces successive occurrences of the regular expression (like the *ed* global substitute command).

index(s, t)
returns the position in string *s* where string *t* first occurs, or 0 if it does not occur at all.

length(s)
returns the length of its argument taken as a string, or of the whole line if there is no argument.

match(s, re)
returns the position in string *s* where the regular expression *re* occurs, or 0 if it does not occur at all. RSTART is set to the starting position (which is the same as the returned value), and RLENGTH is set to the length of the matched string.

split (s,a,s)
splits the string *s* into array elements *a [1], a [2], a [n]*, and returns *n*. The separation is done with the regular expression *fs* or with the field separator FS if *fs* is not given.

sprintf(fmt,expr,expr, . . .)
formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

sub(for,repl,in)
substitutes the string *repl* in place of the first instance of the regular expression *for* in string *in* and returns the number of substitutions. If *in* is omitted, *awk* substitutes in the current record ($0).

substr(s, m, *n)*
returns the *n -character* substring of *s* that begins at position *m* .

The input/output and general functions are:

close(filename)
closes the file or pipe named *filename*.

cmd |getline
pipes the output of *cmd* into *getline*; each successive call to *getline* returns the next line of output from *cmd*.

getline
sets **$0** to the next input record from the current input file.

getline <file
sets **$0** to the next record from *file*.

getline var
sets variable *var* instead.

getline var <file
sets *var* from the next record of *file*.

system(cmd)
executes *cmd* and returns its exit status.

All forms of *getline* return 1 for successful input, 0 for end of file, and –1 for an error.

awk also provides user-defined functions. Such functions may be defined (in the pattern position of a pattern-action statement) as:

```
function name(args,...) { stmts }
func name(args,...) { stmts }
```

Function arguments are passed by value if scalar and by reference if array name. Argument names are local to the function; all other variable names are global. Function calls may be nested and functions may be recursive. The **return** statement may be used to return a value.

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

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

Same, with input fields separated by comma and/or blanks and tabs:

```
BEGIN { FS = ",[ \t]*|[ \t]+" }
      { print $2, $1 }
```

Add up first column, print sum and average:

```
      { s += $1 }
END   { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

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

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Simulate *echo*(1):

```
BEGIN {
      for (i = 1; i < ARGC; i++)
            printf "%s", ARGV[i]
      printf "\n"
      exit
      }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
          { print }
```

command line: **awk –f program n=5 input**

SEE ALSO

grep(1), sed(1).
lex(1), printf(3S) in the *Programmer's Reference Manual.*
Programmer's Guide.

BUGS

Input white space is not preserved on output if fields are involved.
There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the `NULL` string (`""`) to it.

NAME

banner – make posters

SYNOPSIS

banner strings

DESCRIPTION

banner prints its arguments (each up to 10 characters long) in large letters on the standard output.

SEE ALSO

echo(1).

NAME

basename, dirname – deliver portions of pathnames

SYNOPSIS

basename *string* [*suffix*]
dirname *string*

DESCRIPTION

basename deletes any prefix ending in / and the *suffix* (if present in *string*) from *string*, and prints the result on the standard output. It is normally used inside substitution marks (` `) within shell procedures.

dirname delivers all but the last level of the pathname in *string*.

EXAMPLES

The following example, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named **cat** in the current directory:

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

The following example will set the shell variable NAME to **/usr/src/cmd**:

```
NAME= dirname /usr/src/cmd/cat.c
```

SEE ALSO

sh(1).

NAME

bc – arbitrary-precision arithmetic language

SYNOPSIS

bc [**–c**] [**–l**] [*file* ...]

DESCRIPTION

bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The *bc*(1) utility is actually a preprocessor for *dc*(1), which it invokes automatically unless the **–c** option is present. In this case, the *dc* input is sent to the standard output instead. The options are:

–c
Compile only. The output is sent to the standard output.

–l
Argument stands for the name of an arbitrary precision math library.

The syntax for *bc* programs is as follows; L means letter a–z, E means expression, S means statement.

Comments
are enclosed in /* and */.

Names
simple variables: L
array elements: L [E]
The words "ibase", "obase", and "scale"

Other operands
arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
length (E) number of significant decimal digits
scale (E) number of digits right of decimal point
L (E , ... , E)

Operators
+ – * / % ^ (% is remainder; ^ is power)
++ — (prefix and postfix; apply to names)
== <= >= != < >
= =+ =– =* =/ =% =^

Statements

```
E
{ S ; ... ; S }
if ( E ) S
while ( E ) S
for ( E ; E ; E ) S
null statement
break
quit
```

Function definitions

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

Functions in –l math library

s(x)	sine
c(x)	cosine
e(x)	exponential
l(x)	log
a(x)	arctangent
j(n,x)	Bessel function

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or new-lines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. "Auto" variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables, empty square brackets must follow the array name.

EXAMPLE

```
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
        }
}
```

defines a function to compute an approximate value of the exponential function and

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

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

FILES

/usr/lib/lib.b	mathematical library
/usr/bin/dc	desk calculator proper

SEE ALSO

dc(1).

BUGS

The *bc* command does not yet recognize the logical operators, && and ||.
for statement must have all three expressions (E's).
quit is interpreted when read, not when executed.

NAME

bdiff – big diff

SYNOPSIS

bdiff *file1 file2* [**n**] [**–s**]

DESCRIPTION

bdiff is used in a manner analogous to *diff*(1) to find which lines in two files must be changed to bring the files into agreement. Its purpose is to allow processing of files which are too large for *diff*.

The parameters to *bdiff* are:

file1 (file2)
The name of a file to be used. If *file1 (file2)* is –, the standard input is read.

n
The number of line segments. The value of *n* is 3500 by default. If the optional third argument is given and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail.

–s
Specifies that no diagnostics are to be printed by *bdiff* (silent option). Note, however, that this does not suppress possible diagnostic messages from *diff*(1), which *bdiff* calls.

bdiff ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. If both optional arguments are specified, they must appear in the order indicated above.

The output of *bdiff* is exactly that of *diff*, with line numbers adjusted to account for the segmenting of the files (i.e., to make it look as if the files had been processed whole). Note that because of the segmenting of the files, *bdiff* does not necessarily find a smallest sufficient set of file differences.

FILES

/tmp/bd?????

SEE ALSO

diff(1), help(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

bfs – big file scanner

SYNOPSIS

bfs [**–**] *name*

DESCRIPTION

The *bfs* command is (almost) like *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024Kb and 32K lines, with up to 4096 characters, including newline, per line (255 for 16-bit machines). *bfs* is usually more efficient than *ed*(1) for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the **w** command. The optional – suppresses printing of sizes. Input is prompted with * if **P** and a carriage return are typed, as in *ed*(1). Prompting can be turned off again by inputting another **P** and carriage return. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under *ed*(1) are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?: > indicates downward search without wrap-around, and < indicates upward search without wrap-around. There is a slight difference in mark names: only the letters **a** through **z** may be used, and all 26 marks are remembered.

The **e**, **g**, **v**, **k**, **p**, **q**, **w**, =, ! and NULL commands operate as described under *ed*(1). Commands such as ——, +++–, +++=, **–12**, and **+4p** are accepted. Note that **1,10p** and **1,10** will both print the first ten lines. The **f** command only prints the name of the file being scanned; there is no *remembered* file name. The **w** command is independent of output diversion, truncation, or crunching (see the **xo**, **xt** and **xc** commands, below). The following additional commands are available:

xf *file*
Further commands are taken from the named *file*. When an EOF is reached, an interrupt signal is received or an error occurs, reading resumes with the file containing the **xf**. The **xf** commands may be nested to a depth of 10.

xn
List the marks currently in use (marks are set by the **k** command).

xo [*file*]

Further output from the **p** and **NULL** commands is diverted to the named *file*, which, if necessary, is created mode 666 (readable and writable by everyone), unless your *umask* setting (see *umask*(1)) dictates otherwise. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

: *label*

This positions a *label* in a command file. The *label* is terminated by newline, and blanks between the **:** and the start of the *label* are ignored. This command may also be used to insert comments into a command file, since labels need not be referenced.

(**.** , **.**)**xb**/*regular expression*/*label*

A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

1. Either address is not between **1** and **$**.
2. The second address is less than the first.
3. The regular expression does not match at least one line in the specified range, including the first and last lines.

On success, **.** is set to the line matched and a jump is made to *label*. This command is the only one that does not issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. The following command is an unconditional jump:

```
xb/^/ label
```

The **xb** command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe, only a downward jump is possible.

xt *number*

Output from the **p** and **NULL** commands is truncated to at most *number* characters. The initial number is 255.

xv [*digit*] [*spaces*] [*value*]

The variable name is the specified *digit* following the **xv**. The commands **xv5100** or **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. For example, using the above assignments for variables **5** and **6**:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would globally search for the characters **100** and print each line containing a match. To escape the special meaning of **%**, a **** must precede it.

```
g/".*\%[cds]/p
```

could be used to match and list lines containing *printf* of characters, decimal integers, or strings.

Another feature of the **xv** command is that the first line of output from a command can be stored into a variable. The only requirement is that the first character of *value* be an **!**. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable **5**, print it, and increment the variable **6** by one. To escape the special meaning of **!** as the first character of *value*, precede it with a ****.

```
xv7\!date
```

stores the value **!date** into variable **7**.

xbz *label*

xbn *label*

These two commands will test the last saved *return code* from the execution of a command (*!command*) or nonzero value, respectively, to the specified label. The following two examples search for the next five lines containing the string **size**:

```
xv55
: l
/size/
xv5!expr %5 – 1
!if 0%5 != 0 exit 2
xbn l
```

```
xv45
: l
/size/
xv4!expr %4 – 1
!if 0%4 = 0 exit 2
xbz l
```

xc [*switch*]

If *switch* is **1**, output from the **p** and **NULL** commands is crunched; if *switch* is **0** it is not. Without an argument, **xc** reverses *switch*. Initially, *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

SEE ALSO

csplit(1), ed(1), umask(1).

DIAGNOSTICS

? for errors in commands, if prompting is turned off. Self-explanatory error messages when prompting is on.

NAME

bpatch – displays or alters byte content of files

SYNOPSIS

bpatch [<*filename*>]

DESCRIPTION

bpatch allows a user to look at specific areas of a file and to change selected bytes. *bpatch* takes no information from the command line except for the optional name of a file to open initially. *bpatch* displays a prompt (>) and waits for user input. The following commands are recognized:

o *filename*
Open the specified file. If another file is already open, close it.

s *xxxx yyyy*
Display the contents of the file from *xxxx* to *yyyy*, where *xxxx* and *yyyy* are hex values. The starting address is rounded down modulo hex 10, and the ending address is rounded up modulo hex 10. The contents are shown both as hex and printed ASCII. Nonprinting characters are shown as periods in the ASCII display.

c *xxxx zz ...*
Change contents of location *xxxx* to *zz*, where *xxxx* and *zz* are hex numbers. Several values can be changed by entering additional values. Each additional value changes the next sequential byte.

d
Print information about the file, e.g., its type, permissions, size.

q
Exit.

The hex numbers *xxxx* and *yyyy* can represent a full 32-bit address (i.e., eight hex characters). The hex number *zz* represents a byte and should be only two hex characters.

bpatch is insensitive to case of alphabetic characters. The above commands, as well as the characters **A-F** within hex numbers, may be uppercase or lowercase.

NOTE

The prompt has a trailing **NULL** character (> NUL) not visible on a normal terminal.

EXAMPLE

s 0 ff	display bytes 0 thru ff
S 10000 100FF	display bytes 10000 thru 100ff
c 400 7F 31 21	change contents of locations 400-402 to 7F 31 21
o newfile	close the present file and open "newfile"
d	display file information
q	exit the *bpatch* program

DIAGNOSTICS

bpatch gives error messages if it is unable to open a file, or if any commands are entered that would read or write when no file is open.

If an illegal command or no input is entered, *bpatch* displays a brief help message.

If the user has read permission but not write permission on a file, *bpatch* will display a message to this effect at open time. The user will be unable to use the **c** command.

NAME

bru – backup and restore utility

SYNOPSIS

bru *modes* [*control options*] [*selection options*] *files*

DESCRIPTION

bru is a SYSTEM V/88 file system backup utility with significant enhancements over other more common utilities, e.g., *tar*, *cpio*, *volcopy*, and *dd*. Some of *bru*s capabilities include:

- Full or incremental backup with quick and easy restoration of files.
- Multiple physical volumes per archive.
- Data integrity assurance via checksum computation on every archive block.
- Ability to properly save and restore directories, symbolic links, block special files, and character special files.
- Comparison of archives with current directory hierarchy.
- Ability to recover files from corrupted archives or damaged media with minimal data loss.
- No inherent maximum archive buffer size.
- Improved performance through random access archive I/O when available.
- Automatic byte or half word swapping as necessary when reading archives produced on other machines.
- Recognition of file name generation patterns in the same form as the shell for files read from an archive.

When *files* are specified on the command line, the actions to be performed are limited to those *files*. If a named file is a directory, it and all its descendents are used. If no *files* are specified, the default for writing archives is all files in and below the current directory. The default for reading archives is selection of all files in the archive.

If - is given instead of *files*, then the standard input is read to obtain the file list. This is useful with the *find* command to provide finer control over files selected for backup. Obviously, this mode is only valid when *bru* is not also reading its archive from the standard input.

DEFAULTS

Various default parameters, e.g., archive device name and size, archive buffer size, controlling terminal name, etc. are system dependent. These defaults, along with version, variant, and other miscellaneous internal information may be discovered via the **–h** mode.

MODES

One or more of the following modes must be specified. The order of execution, from highest priority to lowest, is **ecitxdgh**.

–c

Create a new archive. Forces a new archive to be created regardless of whether one currently exists. Writing starts at the first block.

–d

Differences between archived *files* and current *files* are detected and reported. May be specified more than once, as **–dd –ddd** or **–dddd** to control level of difference checking.

When specified as **-d**, *bru* reports when it discovers that a regular file's size (st_size) or contents (when compared as byte streams) has changed since the archive was made.

When specified as **-dd**, *bru* reports additional differences in modification date (st_mtime), access mode (st_mode), number of links (st_nlink) for non-directory files, differences in the contents of symbolic links, owner id (st_uid), and group id (st_gid).

When specified as **-ddd**, *bru* reports additional differences in host device (st_dev), major/minor device (st_rdev) for special files, and time of last access (st_atime) for regular files.

When specified as **-dddd**, *bru* reports all differences except time of last status change (st_ctime is not resettable), major/minor device numbers for non-special files (meaningless), and size differences for directory files (may have empty entries). The **–dddd** mode is generally only meaningful during a verification pass with full backups of quiescent file systems.

–e

Estimate media requirements for archive creation with same arguments. Prints estimated number of volumes, number of files to be archived, total number of archive blocks, and total size of archive in kilobytes. If the media size is unknown or unspecified, it is assumed to be infinite.

–g

Dump archive info block in a form more easily parsed by programs implementing a complete file system management package. Performs no other archive actions.

–h

Print **help** summary of options. Also prints some internal information, such as, version number and default values for archive pathname, media size, archive buffer size.

–i

Inspect archive for internal consistency and data integrity. When **–vv** option is also given, prints information from archive header block.

–t

List **table** of contents of archive. When used with the **–v** option, gives a verbose table of contents in the same format as the *ls –l* command. When used with the **-vv** option, also indicates what files are linked to other files and where symbolic links point to.

–x

Extract named *files* from archive. If an archived file is extracted (see **–u** option), then the access mode, device id (special files only), owner uid, group uid, access time, and modification time are also restored. If the **–C** flag is given (see below), the owner uid and group uid will be changed to that of the current user.

Nonexistent directories are recreated from archived directories if possible, otherwise, they are created with appropriate defaults for the current user. Extracted or created directories are initially empty.

CONTROL OPTIONS

Many of the control options are similar in function to their *tar* or *cpio* equivalents.

Sizes are specified in bytes. The scale factors **M**, **k**, or **b** can be used to indicate multiplication by 2**20, 1024, or 512 respectively. Thus "10k", "20b", and "10240" all specify the same number of bytes.

–# str

Use string *str* as a control string for the built-in debugging system. This option provides information about the internal workings of *bru* for the software maintainer or the merely curious. Some examples are given later.

–a

Do not reset the **access** times of disk files that have been read while performing other actions. Normally, *bru* restores the access and modification times of disk files after they have been read. Resetting the times prevents defeat of the mechanism used to track down and remove "dead" files that have not been accessed in any meaningful way recently.

–b *bsize*

Use *bsize* as the archive input/output **buffer** size. The minimum is the size of an archive block (2k or 2048 bytes) and the maximum is determined by available memory and I/O device limitations. If *bsize* is not an even multiple of 2048 bytes, it will be rounded up. Normally this option is only required with the **–c** mode since *bru* writes this information in the archive header block. If specified, *bsize* overrides any existing default value (generally 20k), whether built in or read from the archive header.

–B

Useful in shell scripts where *bru* is run in the **background** with no operator present. Under these conditions, *bru* simply terminates with appropriate error messages and status, rather than attempting interaction with the terminal.

–C

Change the owner (**chown**) and group of each extracted file to the owner uid and group gid of the current user. Normally, *bru* restores the owner and group to those recorded in the archive. This flag causes *bru* to follow the system default, with extracted files having the same owner and group as the user running *bru*, including root.

The

–C option is useful with archives imported from other systems. In general, it should not be used by the operator or System Administrator when restoring saved files. Use the **–tv** option to see the owner and group of files stored in the archive.

–f *path*

Use *path* as the archive **file** instead of the default. If the *path* is "–" then *bru* uses the standard input for archive reading or standard output for archive writing, as appropriate.

–F

Fast mode. In fast mode, checksum computations and comparisons are disabled. This mode is useful when the output of one *bru* is piped to the input of another *bru*, or when the data integrity of the archive transmission medium is essentially perfect. Archives recorded with fast mode enabled must also be read with fast mode.

Also, be aware that some of the automatic features of *bru*, e.g., automatic byte swapping, are not functional in fast mode.

–L *str*
Label the archive with the specified string *str*. *str* is limited to 63 characters and is usually some meaningful reminder pertaining to the archive contents.

–l
Ignore unresolved **links**. Normally, *bru* reports problems with unresolved links (both regular and symbolic links). This option suppresses all such complaints.

–m
Do not cross **mounted** file system boundaries during expansion of explicitly named directories. This option applies only to directories named in *files*. It limits selection of directory descendents to those located on the same file system as the explicitly named directory. This option currently applies only to the **–c** and **–e** modes.

–p
Pass over files in archive by reading rather than seeking. Normally *bru* uses random access capabilities if available. This option forces reading instead of seeks.

–R
Remote files are to be **excluded** from the archive. If the system does not support remote file systems, this option is ignored.

–s *msize*
Use *msize* as the media **size**. The effective media **size** will be computed from *msize* since it must be an integral multiple of the input/output buffer size (see the **–b** option). Normally, this option is only required with the **–c** mode since *bru* writes this information in the archive header block. If specified, *msize* overrides any existing default value, whether built in or read from the archive header.

–v
Enable **verbose** mode. May be specified more than once, as **–vv**, **–vvv**, or **–vvvv**, to get even more verbosity.

–w

Wait for confirmation. *bru* will print the file name, the action to be taken, and **wait** for confirmation. Any response beginning with 'y' will cause the action to complete. Any other response will abort the action.

FILE SELECTION OPTIONS

The file selection options control which files are selected for processing. Note that some options are only valid with specific modes.

–n *date*

Select only files **newer** than *date*. The *date* is given in one of the forms:

DD–MMM–YY[,HH:MM:SS]	EX: 12–Mar–84,12:45:00
MM/DD/YY[,HH:MM:SS]	EX: 3/12/84
MMDDHHMM[YY]	EX: 0312124584
pathname	EX: /etc/lastfullbackup

The time of day is optional in the first two forms. If present, it is separated from the date with a comma.

If *date* is really the pathname of a file, then the modification date of that file will be used instead. This is useful in automated backups when a dummy file is "touched" to save the date of last backup.

–o *user*

Select only files **owned** by *user*. *user* may be specified in one of three ways:

- As an ASCII string corresponding to a user name in the password file.
- As the pathname of a file in which case the owner of that file is used.
- As a numeric value (decimal).

–u *flags*

When used with **–x** mode, causes files of type specified by *flags* to be **unconditionally** selected regardless of modification times. Normally, *bru* will not overwrite (supersede) an existing file with an older archive file of the same name. Files that are not superseded will give warnings if **verbose** mode level 2 (**–vv**) or higher is enabled. Possible characters for *flags* are:

b	select block special files
c	select character special files
d	select directories
l	select symbolic links
p	select fifos (named pipes)
r	select regular files

Selection of directories only implies that their attributes may be modified. Existing directories are never overwritten, this option merely allows their attributes to be set back to some previously existing state.

EXAMPLES

Create (**–c**) a new archive of all files under **/usr/src**, writing archive to file (**–f**) **/dev/rst0** using multiple tapes with a maximum size (**–s**) of 30Mb per tape:

```
bru –c –f /dev/rst0 –s 30M /usr/src
```

Create (**–c**) a new archive on the default device in the first pass, archiving all files in and below the current directory that have been created or modified (**–n**) since 3 P.M. on 14–Jan–84. Then, do a second pass to verify that there are no differences (**–d**) between the archive and current files. Each file is listed (**–v**) as it is processed:

```
bru –cvd –n 14–Jan–84,15:00:00
```

Archive all files owned (**–o**) by user "user1" using the default archive device:

```
find / –user user1 –print | bru –c –
bru –c –o user1 /
```

Copy a directory hierarchy from **/usr/u1** to **/usr/u2**:

```
(cd /usr/u1; bru –cf – ) | (cd /usr/u2; bru –xf –)
```

Extract (**–x**) the regular file **/usr/guest/myfile** unconditionally (**–ur**) from an archive on file (**–f**) **/dev/rf0**. Since the device size was recorded in the header block, it need not be specified. Note that option arguments do not need to be separated from their corresponding option flag by white space:

bru –x –ur –f/dev/rf0 ./usr/guest/myfile

Extract (**–x**) all C source files in **/usr/src/cmd** that have names beginning with characters 'a' through 'm'. Wait (**–w**) for confirmation before extracting each file:

bru –xw '/usr/src/cmd/[a–m]*.c'

Inspect (**–i**) a previously created archive on the default device, dumping the contents of the header block for inspection (**–vvv**) and verifying internal consistency and data integrity of the archive:

bru –ivvv

Perform the same function as the last example except enable various features of the built-in debugger (when linked in). The debug control string is a string of the form **-#<opt1>:<opt2>:...**, where each option is either a single flag character or a flag character followed by a comma separated list. Available flag characters are: **d** enable debugging for list of keywords, **f** limit debugging to list of function names, **F** print source file name, **L** print source file line numbers, **n** print nesting depth, **o** redirect output to listed file, **p** print process name, **t** enable tracing.

bru –ivvv –#t
bru –ivvv –#d:t
bru –ivvv –#d,ar_io,verify:F:L
bru –ivvv –#d:f,ar_seek
bru –ivvv –#d:o,trace.out:t:p

Back up the entire root file system without crossing mounted (**–m**) file system boundaries. The archive will be written to file (**–f**) **/dev/rst0** using an I/O buffer size (**–b**) of 10Kb. A record of all files processed will be written to file **brulogfile** for future reference.

cd /
bru –cvm –f /dev/rst0 –b 10k >brulogfile

DIAGNOSTICS

Most diagnostics are reasonably informative. The most common have to do with meaningless combinations of options, incompatible options, hitting memory or device limits, unresolved file links, trying to archive or restore something to which access is normally denied, or problems with media errors and/or archive corruption.

DEVICE TABLE

bru contains an internal table of known devices and their characteristics. This table is dynamically loaded from a data file specified by the environment variable **BRUTAB**, or from **/etc/brutab**, or from an internal default description if neither of the preceding is found.

SIGNAL HANDLING

bru normally catches both interrupt (SIGINT) and quit (SIGQUIT). When interrupt is caught during archive creation or extraction, *bru* completes its work on the current file before cleaning up and exiting. This is the normal way of aborting *bru*. When a quit signal is caught, an immediate exit is taken.

Note that during file extraction, a quit signal may leave the last file only partially extracted. Similarly, a quit signal during archive writing may leave the archive truncated. When either interrupt or quit is caught at any other time, an immediate exit is taken.

ERROR RECOVERY

When properly configured for a given software/hardware environment, *bru* can recover from most common errors. For example, attempts to use unformatted media are detected, allowing substitution of formatted media. Random blocks in an archive can be deliberately overwritten (corrupted) without affecting *bru*'s ability to recover data from the rest of the archive. When I/O errors are detected, retries are performed automatically. Out of order sequencing on multi-volume archive reads is detected, allowing replacement with the correct volume.

DIRECTORIES

When creating non-incremental archives, *bru* automatically archives all directories necessary to fully restore any file from the archive. During extraction, any required directories that do not already exist are restored from the archive if possible, otherwise, they are created with appropriate defaults for the current user.

The net result is that restoration from incremental archives (which may not contain all necessary directories), or incremental restoration from full archives (which may skip directories needed later), may result in creation of directories with the default attributes.

WILDCARDS

When **reading** archives, *bru* recognizes file name generation patterns in the same format as the shell. This allows greater flexibility in specifying files to be extracted, compared, or listed. As a special extension to shell type expansion, the sense of the match is reversed for patterns that begin with '!'.

Note that the patterns may have to be quoted to prevent expansion by the shell. Also note that patterns are processed independently, without regard for any other patterns that may or may not be present. In particular, **/bin/a* /bin/b*** is equivalent to **/bin/[ab]***, but **/bin/!a* /bin/!b*** is equivalent to **/bin/***, *not* **/bin/![ab]***.

BYTE/WORD SWAPPING

While reading archives produced on other machines, *bru* automatically attempts to perform byte and/or word swapping as necessary.

EXIT CODES

bru always returns meaningful status as follows:

0 Normal exit, no errors or warnings.
1 Warnings (or interrupted).
2 Errors (or quit signal).

SEE ALSO

tar(1), cpio(1).
volcopy(1M), finc(1M), frec(1M), ff(1M) in the *System Administrator's Reference Manual.*

WARNINGS

Pathnames are limited to 127 characters in length.

Implementation differences complicate the algorithms for automatic detection of end of file on devices. The algorithms can be fooled, hence, the **–s** option.

Special files moved to a machine other than their original host will generally be useless and possibly even dangerous.

When extracting files from archives, patterns used to match directories may result in some unnecessary directories being extracted. For example, if the pattern is "a/*/c", and the directory "a/b" is encountered in the archive, the directory file "a/b" will be extracted since it will be needed when (and if) the file "a/b/c" is encountered. When in doubt, use the **–w** option.

In order to be able to efficiently archive needed directories, *bru* builds an image of the directory tree for *files* using dynamically allocated memory. Since there may be at most 5120 characters passed on the command line, it is very unlikely that *bru* will run out of memory while building the tree from command line arguments. This is not true of file lists read from the standard input, particularly on machines with limited address space.

Information about file linkages is also kept in memory. Some linkages may be lost if memory is exhausted.

Since *bru* is owned by **root** and runs with "set user id" to allow it to create directories and special files, it makes every attempt to prevent normal users from archiving or extracting files they would normally not have access to. There may be loopholes. Also note that anyone with physical or electronic access to an archive, and knowledge of the archive structure, can recover any of its contents by writing their own file extraction program.

Directories that have file systems mounted on them will not be properly archived until the file system is unmounted. This is not generally a problem.

Explicitly naming both a directory and one of its descendents will cause the descendent to be archived twice, unless they are on separate file systems and the **–m** flag is used.

Explicitly naming a file more than once is ineffective.

When reading from the raw magnetic tape file (**rmtxxx**), *bru* automatically attempts to adjust the I/O buffer size to match that used to record the archive. Under certain circumstances, it may fail and require help via the **–b** option.

NAME

cal – print calendar

SYNOPSIS

cal [[*month*] *year*]

DESCRIPTION

cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. *year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and the United States.

EXAMPLES

An unusual calendar is printed for September 1752. That is the month 11 days were skipped to make up for lack of leap year adjustments. To see this calendar, type: **cal 9 1752**

BUGS

The year is always considered to start in January even though this is historically naive.

Beware that "cal 83" refers to the early Christian era, not the 20th century.

NAME

calendar – reminder service

SYNOPSIS

calendar [–]

DESCRIPTION

calendar consults the file **calendar** in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates, e.g., "Aug. 24," "august 24," "8/24,", are recognized, but not "24 August" or "24/8". On weekends, "tomorrow" extends through Monday.

When an argument is present, *calendar* does its job for every user who has a file **calendar** in his or her login directory and sends them any positive results by *mail*(1). Normally, this is done daily by operating system facilities.

FILES

/usr/lib/calprog	to figure out today's and tomorrow's dates
/etc/passwd	
/usr/lib/caldir/*	

SEE ALSO

mail(1).

BUGS

Your calendar must be public information for you to get reminder service. *calendar's* extended idea of "tomorrow" does not account for holidays.

NAME

cat – concatenate and print files

SYNOPSIS

cat [**–u**] [**–s**] [**–v** [**–t**] [**–e**]] *file* . . .

DESCRIPTION

cat reads each *file* in sequence and writes it on the standard output. Thus:

cat file

prints **file** on your terminal, and:

cat file1 file2 >file3

concatenates **file1** and **file2**, and writes the results in **file3**.

If no input file is given or if the argument – is encountered, *cat* reads from the standard input file.

The following options apply to *cat*:

–u
The output is not buffered. The default is buffered output.

–s
cat is silent about non-existent files.

–v
Causes non-printing characters (with the exception of tabs, newlines and form-feeds) to be printed visibly. ASCII control characters (octal 000 - 037) are printed as ^*n*, where *n* is the corresponding ASCII character in the range octal 100 - 137 (@, A, B, C, . . ., X, Y, Z, [, \,], ^, and _); the DEL character (octal 0177) is printed **^?**. Other non-printable characters are printed as **M-***x*, where *x* is the ASCII character specified by the low-order seven bits.

When used with the **–v** option, the following options may be used:

–t
Causes tabs to be printed as ^Is and formfeeds to be printed as ^Ls.

–e
Causes a **$** character to be printed at the end of each line (before the newline).

The **–t** and **–e** options are ignored if the **–v** option is not specified.

WARNING

Redirecting the output of **cat** onto one of the files being read will cause the loss of the data originally in the file being read. For example, typing:

cat file1 file2 >file1

will cause the original data in **file1** to be lost.

SEE ALSO

cp(1), pg(1), pr(1).

NAME

cb – C program beautifier

SYNOPSIS

cb [**–s**] [**–j**] [**–l** *leng*] [*file* ...]

DESCRIPTION

The *cb* command reads C programs either from its arguments or from the standard input, and writes them on the standard output with spacing and indentation that display the structure of the code. Under default options, *cb* preserves all user newlines.

cb accepts the following options.

–s
Canonicalizes the code to the style of Kernighan and Ritchie in *The C Programming Language*.

–j
Causes split lines to be put back together.

–l *leng*
Causes *cb* to split lines that are longer than *leng*.

SEE ALSO

cc(1).
The C Programming Language. Prentice-Hall, 1978.

BUGS

Punctuation that is hidden in preprocessor statements causes indentation errors.

NAME

cc – C compiler

SYNOPSIS

cc [*options*] ... *files*

DESCRIPTION

The *cc* command is the C compiler. It generates assembly instructions. The following types of arguments are accepted by *cc*:

Arguments whose names end with **.c** are taken to be C source programs; they are compiled and each object program is left on the file whose name is that of the source with **.o** substituted for **.c.** The **.o** file is normally deleted if a single C program is compiled and loaded all at one time. In the same way, arguments whose names end with **.s** are taken to be assembly source programs and are assembled to produce a **.o** file.

The *cc* command presently uses the Green Hills 1.8.4 C compiler. Some of the compiler options described below are specific to that compiler and may not be useable with other C compilers.

The following flags are interpreted by cc. See *ld*(1) for link editor options and *as*(1) for assembler options.

–B *string*
Construct pathnames for substitute preprocessor, compiler, assembler, and link editor passes by concatenating *string* with the suffixes **cpp, comp, reorder, as,** and **ld.** If *string* is empty, it is taken to be **/lib** for all programs except **as** and **ld,** for which it is **/bin.**

–c
Suppress the link-editing phase of the compilation, and force an object file to be produced.

–D *symbol*
Define *symbol* to the preprocessor. This mechanism is useful with the conditional statements in the preprocessor by allowing symbols to be defined external to the source file.

–E
Run only *cpp*(1) on the named C programs, and send the result to the standard output.

–I *dir*
Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in double quotes are searched for first in the directory of the *file* argument, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.

–O
Invoke the object code optimizer. The Green Hills C compiler allows two further levels of optimization control through the OPTIM environment variable. (See below).

–P
Run only *cpp*(1) on the named C programs; leave the result in corresponding files suffixed **.i.**

–S
Compile the named C programs and leave the assembler-language output in corresponding files suffixed .s.

–U *symbol*
Undefine *symbol* to the preprocessor.

–W*c,arg1[,arg2...]*
Hand off the argument(s) *argi* to pass *c*, where *c* is one of **[p02al]** indicating preprocessor, compiler, reorder, assembler, or link editor, respectively. For example: **–Wa,-m** invokes the *m4*(1) macro preprocessor on the input to the assembler. This must be done for a source file that contains assembler escapes.

–Xnnn
Turn ON the Green Hills option number *nnn* in the compiler pass. The set of Green Hills options that may be used are documented in the *Software Generation System Guide* section in the *Programmer's Guide.*

–Y[p02alSILU],*dirname*

Specify a new pathname, *dirname*, for the locations of the tools and directories designated by the first argument. **[p02alSILU]** represents:

p	preprocessor (cpp)
0	compiler (comp)
2	optimizer (reorder)
a	assembler (as)
l	linker (ld)
S	directory that contains the startup routines
I	default directory search by *cpp*(1)
L	first default library searched by *ld*(1)
U	second default library searched by *ld*(1)

–Znnn

Turn OFF the Green Hills option number *nnn* in the compiler pass. This has the opposite effect to the **-X** option.

OPTIMIZATION OPTIONS

The *cc* command will enable different levels of optimization depending on the contents of the OPTIM environment variable. The Green Hills documented options **-OL** and **-OLM** will not work as described. The **L** and **M** options need to be placed in the OPTIM environment variable in order to work correctly. For example, to compile a program with the Green Hills options **–OLM** the following command sequence should be used:

```
$ OPTIM=LM;export OPTIM
$ cc -O example.c
```

Consult the Green Hills documentation for details on the effect of using the **L** and **M** options.

FILES

file.c	input file
file.o	object file
file.s	assembly language file
a.out	link-edited file
/usr/tmp/m88?	temporary
LIBDIR/cpp	preprocessor
LIBDIR/comp	compiler pass 1
LIBDIR/reorder	object code reorder
BINDIR/cc	compiler driver

BINDIR/as	assembler
BINDIR/ld	link editor **LIBDIR/libc.a** runtime library

SEE ALSO

as(1).
ld(1).
The C Programming Language by B.W. Kernighan and D. M. Ritchie, Prentice-Hall, 1978.
Programming in C - A Tutorial by B. W. Kernighan.
C Reference Manual by D. M. Ritchie.
The C Programming Language in the *Software Generation System Guide.*

DIAGNOSTICS

The diagnostics produced by the C compiler are sometimes cryptic. Occasional messages may be produced by the assembler or link editor.

NAME

cd – change working directory

SYNOPSIS

cd [*directory*]

DESCRIPTION

If *directory* is not specified, the value of shell parameter $HOME is used as the new working directory. If *directory* specifies a complete path starting with /, ., .., *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the $CDPATH shell variable. $CDPATH has the same syntax as, and similar semantics to, the $PATH shell variable. *cd* must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *cd* would be ineffective if it were written as a normal command; therefore, it is recognized and is internal to the shell.

SEE ALSO

pwd(1), sh(1).
chdir(2) in the *Programmer's Reference Manual.*

NAME

cdc – change the delta commentary of an SCCS delta

SYNOPSIS

cdc **–r***SID* [**–m**[*mrlist*]] [**–y**[*comment*]] *files*

DESCRIPTION

cdc changes the *delta commentary*, for the SID (**SCCS ID**entification string) specified by the **–r** keyletter, of each named SCCS file.

delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the *delta*(1) command (**–m** and **–y** keyletters).

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If a name of **–** is given, the standard input is read (see *WARNINGS*) and each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of *keyletter* arguments and file names.

All the described *keyletter* arguments apply independently to each named file:

–r*SID*

Used to specify the *S*CCS *ID*entification (SID) string of a delta for which the delta commentary is to be changed.

–m*mrlist*

If the SCCS file has the **v** flag set (see *admin*(1)) then a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the **–r** keyletter *may* be supplied. A **NULL** MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of *delta*(1). To delete an MR, precede the MR number with the character **!** (see **EXAMPLES**). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If **–m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see –y keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the **v** flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *cdc* terminates and the delta commentary remains unchanged.

–y*[comment]*

Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the **–r** keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A NULL *comment* has no effect.

If **–y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the *comment* text.

Simply stated, the keyletter arguments are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

cdc –r1.6 –m"bl78-12345 !bl77-54321 bl79-00001" –ytrouble s.file

Adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment **trouble** to delta 1.6 of s.file.

cdc –r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble

does the same thing.

WARNINGS

If SCCS file names are supplied to the *cdc* command via the standard input (– on the command line), then the **–m** and **–y** keyletters must also be used.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

cflow – generate C flowgraph

SYNOPSIS

cflow [–r] [–ix] [–i_] [–d *num*] *files*

DESCRIPTION

The *cflow* command analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed with **.y**, **.l**, and **.c** are yacced, lexed, and C-preprocessed as appropriate. The results of the preprocessed files, and files suffixed with **.i**, are then run through the first pass of *lint*(1). Files suffixed with **.s** are assembled. Assembled files, and files suffixed with **.o**, have information extracted from their symbol tables. The results are collected and turned into a graph of external references which is displayed upon the standard output.

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol followed by a colon and its definition. Normally only function names that do not begin with an underscore are listed (see the **–i** options below). For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char** *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only < > is printed.

As an example, given the following in *file.c*:

```
int    i;

main()
{
        f();
        g();
        f();
}

f()
{
        i = h();
}
```

The command:

cflow –ix file.c

Produces the output:

```
1    main: int(), <file.c 4>
2        f: int(), <file.c 11>
3            h: <>
4            i: int, <file.c 1>
5        g: <>
```

When the nesting level becomes too deep, the output of *cflow* can be piped to *pr*(1), using the **–e** option, to compress the tab expansion to something less than every eight spaces.

In addition to the **–D**, **–I**, and **–U** options (which are interpreted just as they are by *cc*(1) and *cpp*(1)), the following options are interpreted by *cflow*:

–r
Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

–ix
Include external and static data symbols. The default is to include only functions in the flowgraph.

–i_
Include names that begin with an underscore. The default is to exclude these functions (and data if *–ix* is used).

–d*num*
The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default, this is a very large number. Attempts to set the cutoff depth to a nonpositive integer will be ignored.

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), yacc(1).
pr(1) in the *User's Reference Manual.*

BUGS

Files produced by *lex*(1) and *yacc*(1) cause the reordering of line number declarations which can confuse *cflow*. To get proper results, feed *cflow* the *yacc* or *lex* input.

NAME

chk – file system check and interactive repair

SYNOPSIS

chk [*options*] [*alias*]

DESCRIPTION

chk checks to see that *alias* is in the *permissions* file and that file system check permission is given. Read-only check permission is indicated by an **R** only in the "perms" field of the *permissions* file; read/write permission is indicated by a **W**. *chk* takes the same options as *fsck*(1M).

If neither options nor *alias* is given, *chk* assumes the default alias, **floppy**.

FILES

/etc/fsck
/etc/filesys permissions file

SEE ALSO

fsck(1M) in the *System Administrator's Reference Manual.*
filesys(4) in the *Programmer's Reference Manual.*

NAME

chmod – change mode

SYNOPSIS

chmod *mode file ...*

chmod *mode directory ...*

DESCRIPTION

The permissions of the named *files* or *directories* are changed according to **mode**, which may be symbolic or absolute. Absolute changes to permissions are stated using octal numbers:

chmod *nnn file(s)*

where *n* is a number from 0 to 7. Symbolic changes are stated using mnemonic characters:

chmod *a operator b file(s)*

where *a* is one or more characters corresponding to **user**, **group**, or **other**; where *operator* is +, −, and =, signifying assignment of permissions; and where *b* is one or more characters corresponding to type of permission.

An absolute mode is given as an octal number constructed from the OR of the following modes:

4000	set user ID on execution
20#0	set group ID on execution if # is **7**, **5**, **3**, or **1** enable mandatory locking if # is **6**, **4**, **2**, or **0**
1000	sticky bit is turned on (see *chmod*(2))
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

Symbolic changes are stated using letters that correspond both to access classes and to the individual permissions themselves. Permissions to a file may vary depending on your UID or GID. Permissions are described in three sequences each having three characters:

User	Group	Other
rwx	rwx	rwx

This example (meaning that **u**ser, **g**roup, and **o**thers all have **r**eading, **w**riting, and **e**xecution permission to a given file) demonstrates two categories for granting permissions: the access class and the permissions themselves.

Thus, to change the mode of a file's (or directory's) permissions using *chmod*'s symbolic method, use the following syntax for mode:

[*who*] *operator* [*permission(s)*], ...

A command line using the symbolic method would appear as:

chmod g+rw *file*

This command would make *file* readable and writable by the group.

The *who* part can be stated as one or more of the following letters:

u	**u**ser's permissions
g	**g**roup's permissions
o	**o**thers permissions

The letter **a** (**a**ll) is equivalent to **ugo** and is the default if *who* is omitted.

operator can be + to add *permission* to the file's mode, – to take away *permission*, or = to assign *permission* absolutely. (Unlike other symbolic operations, = has an absolute effect in that it resets all other bits.) Omitting *permission* is only useful with = to take away all permissions.

permission is any compatible combination of the following letters:

r	**r**eading permission
w	**w**riting permission
x	e**x**ecution permission
s	user or group set-ID is turned on
t	sticky bit is turned on
l	mandatory **l**ocking will occur during access

Multiple symbolic modes separated by commas may be given, though no spaces may intervene between these modes. Operations are performed in the order given. Multiple symbolic letters following a single operator cause the corresponding operations to be performed simultaneously. The letter **s** is only meaningful with **u** or **g**, and **t** only works with **u**.

Mandatory file and record locking (**l**) refers to a file's ability to have its reading or writing permissions locked while a program is accessing that file. It is not possible to permit group execution and enable a file to be locked on execution at the same time. In addition, it is not possible to turn on the set-group-ID and enable a file to be locked on execution at the same time. The following examples, are illegal usages and will elicit error messages:

chmod g+x,+l *file*

chmod g+s,+l *file*

As long as the user has the appropriate privilege, if the file is a symbolic link, the file pointed to by the symbolic link (not the symbolic-link file itself) is affected.

Only the owner of a file or directory (or the superuser) may change a file's mode. Only the superuser may set the sticky bit on a non-directory file. If you are not superuser, **chmod** will mask the sticky-bit but will not return an error. In order to turn on a file's set-group-ID, your own group ID must correspond to the file's and group execution must be set.

EXAMPLES

chmod a–x *file*

chmod 444 *file*

The first examples deny execution permission to all. The absolute (octal) example permits only reading permissions.

chmod go+rw *file*

chmod 066 *file*

These examples make a file readable and writable by the group and others.

chmod +l *file*

This causes a file to be locked during access.

chmod =rwx,g+s *file*

chmod 2777 *file*

These last two examples enable all to read, write, and execute the file; and they turn on the set group-ID.

NOTES

In a RFS environment, you may not have the permissions that the output of the **ls –l** command leads you to believe. For more information, see the *Mapping Remote Users* section of Chapter 10 of the *System Administrator's Guide*.

SEE ALSO

ls(1).
chmod(2) in the *Programmer's Reference Manual*.

NAME

chown, chgrp – change owner or group

SYNOPSIS

chown *owner file ...*

chown *owner directory ...*

chgrp *group file ...*

chgrp *group directory ...*

DESCRIPTION

The *chown* command changes the owner of the *files* or *directories* to *owner*. The owner may be either a decimal user ID or a login name found in the password file.

The *chgrp* command changes the group ID of the *files* or *directories* to *group*. The group may be either a decimal group ID or a group name found in the group file.

If either command is invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode (04000 and 02000, respectively) are cleared.

Only the owner of a file (or the superuser) may change the owner or group of that file. If _POSIX_CHOWN_RESTRICTED is in effect (system configuration option, off by default), only the superuser may change the owner of a file. A user invoking *chgrp* must belong to the specified group and be the owner of the file or be the superuser.

If *file* is a symbolic link, the link itself is changed, not the file to which it points.

FILES

/etc/passwd
/etc/group

NOTES

In a RFS environment, you may not have the permissions that the output of the **ls –l** command leads you to believe. For more information, see the *Mapping Remote Users* section of Chapter 10 of the *System Administrator's Guide.*

SEE ALSO

chmod(1).
chown(2), group(4), passwd(4) in the *Programmer's Reference Manual.*

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] *file1 file2*

DESCRIPTION

The two files are compared. (If *file1* is **–**, the standard input is used.) Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

Options:

–l
Print the byte number (decimal) and the differing bytes (octal) for each difference.

–s
Print nothing for differing files; return codes only.

SEE ALSO

comm(1), diff(1).

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

col – filter reverse linefeeds

SYNOPSIS

col [–b] [–f] [–x] [–p]

DESCRIPTION

col reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse line feeds (ASCII code ESC-7), and by forward and reverse half-linefeeds (ESC-9 and ESC-8). *col* is particularly useful for filtering multicolumn output made with the **.rt** command of *nroff* and output resulting from use of the *tbl*(1) preprocessor.

If the –b option is given, *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.

Although *col* accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. This treatment can be suppressed by the –f (fine) option; in this case, the output from *col* may contain forward half-linefeeds (ESC-9), but will still never contain either kind of reverse line motion.

Unless the –x option is given, *col* will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters SO (\017) and SI (\016) 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.

On input, the only control characters accepted are space, backspace, tab, return, newline, SI, SO, VT (\013), and ESC followed by 7, 8, or 9. The VT character is an alternate form of full reverse linefeed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

Normally, *col* will ignore any escape sequences unknown to it that are found in its input; the –p option may be used to cause *col* to output these sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless the user is fully aware of the textual position of the escape sequences.

NOTES

The input format accepted by *col* matches the output produced by *nroff* with either the **–T37** or **–Tlp** options. Use **–T37** (and the **–f** option of *col*) if the ultimate disposition of the output of *col* will be a device that can interpret half-line motions, otherwise **–Tlp**.

BUGS

Cannot back up more than 128 lines.

Allows at most 800 characters, including backspaces, on a line.

Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NAME

comb – combine SCCS deltas

SYNOPSIS

comb *files*

DESCRIPTION

comb generates a shell procedure (see *sh*(1)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored.

If a name of – is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

The following describes keyletter arguments; each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file:

-p *SID*
The SID of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-c *list*
A *list* of deltas to be preserved. (See *get*(1) for the syntax of a *list*.) All other deltas are discarded.

-o
For each **get –e** generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the **–o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-s

This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, determine exactly how much space is saved by the combining process.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

EXAMPLE

comb s.file1 > tmp1

produces a shell script saved in **tmp1** that will remove from the SCCS-format file **s.file1** all deltas previous to the last set of changes, i.e., removes the capability of returning to earlier versions.

FILES

COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

admin(1), delta(1), get(1), prs(1), sccsfile(4).
help(1), sh(1) in the *User's Reference Manual*.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

comm – select or reject lines common to two sorted files

SYNOPSIS

comm [– [**123**]] *file1 file2*

DESCRIPTION

comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence (see *sort*(1)), and produces a three-column output: lines only in *file1*; lines only in *file2*; and lines in both files. The file name – means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus, **comm –12** prints only the lines common to the two files; **comm –23** prints only lines in the first file but not in the second; **comm –123** prints nothing.

SEE ALSO

cmp(1), diff(1), sort(1), uniq(1).

NAME

conv – common object file converter

SYNOPSIS

conv [**–a**] [**–o**] [**–p**] **–t** *target* [**–** | *files*]

DESCRIPTION

The *conv* command converts object files in the common object file format from their current byte ordering to the byte ordering of the *target* machine. The converted file is written to **file.v**. The *conv* command can be used on either the source (sending) or target (receiving) machine.

Command line options are:

–
indicates that the names of *files* should be read from the standard input.

–a
If the input file is an archive, produce the output file in the UNIX System V Release 2.0 portable archive format.

–o
If the input file is an archive, produce the output file in the old (pre-UNIX System V) archive format.

–p
If the input file is an archive, produce the output file in the UNIX System V Release 1.0 random access archive format.

–t *target*
Convert the object file to the byte ordering of the machine (*target*) to which the object file is being shipped. This may be another host or a target machine. Legal values for *target* are: pdp, vax, ibm, x86, b16, n3b, mc68 and m32.

The *conv* command is meant to ease the problems created by a multi-host cross-compilation development environment. The *conv* command is best used within a procedure for shipping object files from one machine to another.

The *conv* command will recognize and produce archive files in three formats: the pre-UNIX System V format, the UNIX System V Release 1.0 random access format, and the UNIX System V Release 2.0 portable ASCII format. By default, *conv* will create the output archive file in the same format as the input file. To produce an output file in a different format than

the input file, use the **–a**, **–o**, or **–p** option. If the output archive format is the same as the input format, the archive symbol table will be converted, otherwise the symbol table will be stripped from the archive. The *ar*(1) command with its **–t** and **–s** options must be used on the target machine to recreate the archive symbol table.

DIAGNOSTICS

The diagnostics are self-explanatory. Fatal diagnostics on the command lines cause termination. Fatal diagnostics on an input file cause the program to continue to the next input file.

CAVEATS

The *conv* command will not convert archives from one format to another if both the source and target machines have the same byte ordering. You should use *convert*(1) for this purpose.

SEE ALSO

ar(1), convert(1), ar(4), a.out(4).

NAME

convert – convert archive files to common formats

SYNOPSIS

convert *infile outfile*

DESCRIPTION

The *convert* command transforms input *infile* to output *outfile*. *infile* must be a UNIX System V Release 1.0 archive file and *outfile* will be the equivalent UNIX System V Release 2.0 archive file. All other types of input to the *convert* command will be passed unmodified from the input file to the output file (along with appropriate warning messages).

infile must be different from *outfile*.

FILES

TMPDIR/conv* temporary files

TMPDIR is usually **/usr/tmp** but can be redefined by setting the environment variable TMPDIR (see *tempnam*() in *tmpnam*(3S)).

SEE ALSO

ar(1), tmpnam(3S), a.out(4), ar(4)

NAME

cp, ln, mv – copy, link or move files

SYNOPSIS

cp *file1* [*file2* ...] *target*
ln [**–f**] [**–s**] *file1* [*file2* ...] *target*
mv [**–f**] *file1* [*file2* ...] *target*

DESCRIPTION

file1 is copied (linked, moved) to *target*. Under no circumstance can **file1** and *target* be the same (take care when using *sh*(1) metacharacters). If *target* is a directory, then one or more files are copied (linked, moved) to that directory. If *target* is a file, its contents are destroyed.

If *mv* or *ln* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)), ask for a response, and read the standard input for one line; if the line begins with **y**, the *mv* or *ln* occurs, if permissible; if not, the command exits. For *mv*, when the parent directory of **file1** is writable and has the sticky bit set, one or more of the following conditions must be true:

The user must own the file
The user must own the directory
The file must be writable by the user
The user must be the superuser

When the **–f** option is used or if the standard input is not a terminal, no questions are asked and the *mv* or *ln* is done.

Only *mv* will allow **file1** to be a directory, in which case, the directory rename will occur only if the two directories have the same parent; **file1** is renamed *target*. If **file1** is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file.

When using *cp*, if *target* is not a file, a new file is created that has the same mode as **file1** except that the sticky bit is not set unless you are superuser; the owner and group of *target* are those of the user. If *target* is a file, copying a file into *target* does not change its mode, owner, nor group. The last modification time of *target* (and last access time, if *target* did not exist) and the last access time of *file1* are set to the time the copy was made. If *target* is a link to a file, all links remain and the file is changed.

There are two kinds of links: hard links and symbolic links. By default, *ln* makes hard links. A hard link to a file is indistinguishable from the original directory entry; any changes to a file are effective independent of the name used to reference the file. Hard links may not span file systems and may not refer to directories.

The **-s** option causes *ln* to create symbolic links. A symbolic link contains the name of the file to which it is linked. The referenced file is used when an *open*(2) is performed on the link. A *stat*(2) on a symbolic link will return the linked-to file. An *lstat*(2) must be done to obtain information about the link. The *readlink*(2) call may be used to read the contents of a symbolic link. Symbolic links may span file systems and may refer to directories.

SEE ALSO

chmod(1), cpio(1), lstat(2), readlink(2), rm(1), symlink(2).

WARNINGS

ln will not link across file systems. This restriction is necessary because file systems can be added and removed.

BUGS

If **file1** and *target* lie on different file systems, *mv* must copy the file and delete the original. In this case, any linking relationship with other files is lost.

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio –o [**acBvV**] [**–C** *bufsize*] [[**–O** *file*] [**–M** *message*]]

cpio –i [**BcdmrtuvVfsSb6k**] [**–C** *bufsize*] [[**–I** *file*] [**–M** *message*]] [*patterns* ...]

cpio –p [**adlLLmuvV**] *directory*

DESCRIPTION

cpio **–o** (copy out) reads the standard input to obtain a list of pathnames and copies those files onto the standard output together with pathname and status information. Output is padded to a 512-byte boundary by default.

cpio **–i** (copy in) extracts files from the standard input, which is assumed to be the product of a previous **cpio –o**. Only files with names that match *patterns* are selected. *patterns* are regular expressions given in the filename-generating notation of *sh*(1). In *patterns*, metacharacters **?**, *****, and **[...]** match the slash (/) character, and backslash (\) is an escape character. A **!** metacharacter means *not*. (For example, the **!abc*** pattern would exclude all files that begin with **abc**.) Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is ***** (i.e., select all files). Each *pattern* must be enclosed in double quotes, otherwise, the name of a file in the current directory is used.

Extracted files are conditionally created and copied into the current directory tree based upon the options described below. The permissions of the files will be those of the previous *cpio* **–o**. The owner and group of the files will be that of the current user unless the user is superuser, which causes *cpio* to retain the owner and group of the files of the previous cpio **–o**.

NOTE:

If **cpio –i** tries to create a file that already exists and the existing file is the same age or newer, *cpio* will output a warning message and not replace the file. (The **–u** option can be used to unconditionally overwrite the existing file.)

cpio **–p** (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination *directory* tree based upon the options described below.

The meanings of the available options are

–a
Reset access times of input files after they have been copied. Access times are not reset for linked files when **cpio –pla** is specified.

–b
Reverse the order of the bytes within each word. Use only with the **–i** option.

–B
Input/output is to be blocked 5,120 bytes to the record. The default buffer size is 512 bytes when this and the **C** options are not used. (**–B** does not apply to the pass option; **–B** is meaningful only with data directed to or from a character special device, e.g., **/dev/rmt/ctape**.)

–c
Write header information in ASCII character form for portability. Always use this option when origin and destination machines are different types.

–C *bufsize*
Input/output is to be blocked *bufsize* bytes to the record, where *bufsize* is replaced by a positive integer. The default buffer size is 512 bytes when this and **B** options are not used. (**–C** does not apply to the *pass* option; **–C** is meaningful only with data directed to or from a character special device, e.g., **/dev/rmt/ctape**.)

–d
directories are to be created as needed.

–f
Copy in all *files* except those in *patterns*. (See the paragraph on cpio –i for a description of *patterns*.)

–I *file*
Read the contents of *file* as input. If *file* is a character special device, when the first medium is full, replace the medium and type a carriage return to continue to the next medium. Use only with the **-i** option.

–k

Attempt to skip corrupted file headers and I/O errors that may be encountered. If you want to copy files from a medium that is corrupted or out of sequence, this option lets you read only those files with good headers. (For *cpio* archives that contain other *cpio* archives, if an error is encountered *cpio* may terminate prematurely. *cpio* will find the next good header, which may be one for a smaller archive, and terminate when the smaller archive's trailer is encountered.) Used only with the **-i** option.

–l

Whenever possible, link files rather than copying them. Usable only with the **–p** option.

–L

Follow symbolic links; the default is not to follow links. If an archive is made from a tree containing symbolic links, it will record the path associated with each link. When it is restored, the symbolic links will be re-made. If **–L** is specified, the actual file pointed to by the link is archived instead of the symbolic link contents.

–m

Retain previous file modification time. This option is ineffective on directories that are being copied.

–M *message*

Define a message to use when switching media. When you use the **–O** or **–I** options and specify a character special device, you can use this option to define the message that is printed when you reach the end of the medium. One **%d** can be placed in the message to print the sequence number of the next medium needed to continue.

–O *file*

Direct the output of *cpio* to *file*. If *file* is a character special device, when the first medium is full, replace the medium and type a carriage return to continue to the next medium. Use only with the **–o** option.

–r

Interactively rename files. If the user types a NULL line, the file is skipped. If the user types a ".", the original pathname is copied. (Not available with *cpio* **–p**.)

–s
Swap bytes within each half word. Use only with the **–i** option.

–S
Swap halfwords within each word. Use only with the **–i** option.

–t
Print a table of contents of the input. No files are created.

–u
Copy unconditionally. (Normally, an older file will not replace a newer file with the same name.)

–v
(Verbose) causes a list of file names to be printed. When used with the **–t** option, the table of contents looks like the output of an *ls* **–l** command (see **ls**(1)).

–V
(Special verbose) prints a dot for each file seen. Useful to assure the user that *cpio* is working without printing out all file names.

–6
Process an old (i.e., UNIX System Sixth Edition format) file. Use only with the **–i** option.

NOTE: *cpio* assumes four-byte words.

If *cpio* reaches the end of the medium (the end of a cartridge tape for example) when writing to (**–o**) or reading from (**–i**) a character special device and **–O** and **–I** aren't used, *cpio* will print the message:

```
If you want to go on, type device/file name when ready.
```

To continue, you must replace the medium and type the character special device name e.g., (**/dev/rmt/ctape**) and carriage return. You may want to continue by directing *cpio* to use a different device. For example, if you have two tape drives you may want to switch between them so *cpio* can proceed while you are changing the tape. A carriage return alone causes the *cpio* process to prompt the user if he/she really wants to exit. If the user answers yes, then **cpio** will exit, otherwise the message:

```
If you want to go on, type device/file name when ready.
```

displays again. The reason for this is to prevent users from accidently exiting *cpio* should the terminal they are using have a screen saver feature, and they press **RETURN** to turn the screen on again.

NAME

cpp – the C language preprocessor

SYNOPSIS

LIBDIR/*cpp* [*option* ...] [*ifile* [*ofile*]]

DESCRIPTION

The C language preprocessor, *cpp*, is invoked as the first pass of any C compilation by the *cc*(1) command. Thus, *cpps* output is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, *cpp* and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of *cpp* other than through the *cc*(1) command is not suggested, since the functionality of *cpp* may someday be moved elsewhere. See *m4*(1) for a general macro processor.

cpp optionally accepts two file names as arguments. *ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

–P
Preprocess the input without producing the line control information used by the next pass of the C compiler.

–C
By default, *cpp* strips C-style comments. If the –C option is specified, all comments (except those found on *cpp* directive lines) are passed along.

–U*name*
Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. Following is the current list of these possibly reserved symbols. In SYSTEM V/88, *sysV88*, *unix* and *m88k* are defined.

Operating system:	unix, sysV88, dmert, gcos, ibm, os, tss
Hardware:	interdata, pdp11, u370, u3b, u3b5, u3b2, u3b20d, vax, m88k
System variant:	RES, RT
lint(1):	lint

–D*name*

–D*name*=*def*

Define *name* with value *def* as if by a **#define**. If no =*def* is given, *name* is defined with value 1. The **–D** option has lower precedence than the **–U** option, i.e., if the same name is used in both a **–U** option and a **–D** option, the name will be undefined regardless of the order of the options.

–T

The **–T** option forces *cpp* to use only the first eight characters to distinguish preprocessor symbols and is included for backward compatibility.

–I*dir*

Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" will be searched for first in the directory of the file with the **#include** line, then in directories named in **–I** options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the file with the **#include** line is not searched.

–Y*dir*

Use directory *dir* in place of the standard list of directories when searching for **#include** files.

–H

Print, one per line on standard error, the pathnames of included files.

Two special names are understood by *cpp*. The name **__LINE__** is defined as the current line number (as a decimal integer) as known by *cpp*, and **__FILE__** is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directive lines start with # in column 1. Any number of blanks and tabs is allowed between the # and the directive. The directives are:

#define *name token-string*

Replace subsequent instances of *name* with *token-string*.

#define *name*(*arg*, ..., *arg*) *token-string*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma-separated sets of tokens, and a) followed by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, *cpp* re-starts its scan for names to expand at the beginning of the newly created *token-string*.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on. No additional tokens are permitted on the directive line after *name.*

#ident "*string*"
Put *string* into the .comment section of an object file.

#include "*filename*"

#include <*filename*>
Include at this point the contents of *filename* (which will then be run through *cpp*). When the <*filename*> notation is used, *filename* is only searched for in the standard places. See the **–I** and **–Y** options above for more detail. No additional tokens are permitted on the directive line after the final " or >.

#line *integer-constant* "*filename*"
Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file from which it comes. If "*filename*" is not given, the current file name is unchanged. No additional tokens are permitted on the directive line after the optional *filename.*

#endif
Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**. No additional tokens are permitted on the directive line.

#ifdef *name*
The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**. No additional tokens are permitted on the directive line after *name*.

#ifndef *name*

The lines following will appear in the output if and only if *name* has not been the subject of a previous **#define**. No additional tokens are permitted on the directive line after *name*.

#if *constant-expression*

Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, the unary **−**, **!**, and **~** operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined (** *name* **)** or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these operators, integer constants, and names which are known by *cpp* should be used in *constant-expression*. In particular, the **sizeof** operator is not available.

To test whether either of two symbols, *foo* and *fum*, are defined, use

```
#if defined(foo) | defined(fum)
```

#elif *constant-expression*

An arbitrary number of **#elif** directives is allowed between a **#if**, **#ifdef**, or **#ifndef** directive and a **#else** or **#endif** directive. The lines following the **#elif** directive will appear in the output if and only if the preceding test directive evaluates to zero, all intervening **#elif** directives evaluate to zero, and the *constant-expression* evaluates to non-zero. If *constant-expression* evaluates to non-zero, all succeeding **#elif** and **#else** directives will be ignored. Any *constant-expression* allowed in a **#if** directive is allowed in a **#elif** directive.

#else

The lines following will appear in the output if and only if the preceding test directive evaluates to zero, and all intervening **#elif** directives evaluate to zero. No additional tokens are permitted on the directive line.

The test directives and the possible **#else** directives can be nested.

FILES

INCDIR	standard directory list for **#include** files, usually **/usr/include**
LIBDIR	usually **/lib**

SEE ALSO

cc(1), lint(1), m4(1).

DIAGNOSTICS

The error messages produced by *cpp* are intended to be self-explanatory. The line number and file name where the error occurred are printed along with the diagnostic.

NOTES

The unsupported **–W** option enables the **#class** directive. If it encounters a **#class** directive, *cpp* will exit with code 27 after finishing all other processing. This option provides support for "C with classes".

Because the standard directory for included files may be different in different environments, this form of **#include** directive:

```
#include <file.h>
```

should be used, rather than one with an absolute path, like:

```
#include "/usr/include/file.h"
```

cpp warns about the use of the absolute pathname.

NAME

cprs – compress a common object file

SYNOPSIS

cprs [**–p**] *file1 file2*

DESCRIPTION

The *cprs* command reduces the size of a common object file, **file1**, by removing duplicate structure and union descriptors. The reduced file, **file2**, is produced as output.

The sole option to *cprs* is:

–p
Print statistical messages including: total number of tags, total duplicate tags, and total reduction of **file1**.

SEE ALSO

strip(1), a.out(4), syms(4).

NAME

crc - generate cyclic redundancy checksums (crc) of files

SYNOPSIS

crc [-frcld] - | *file_list*

DESCRIPTION

The *crc* shell command utility is a versatile tool for use in generating 16-bit *crc* values of an input stream. The input stream can consist either of data or of names of files to be checked. There are four different display options available.

If the file to be checked is an object file, *crc* will ignore the compiler-generated time stamps embedded in the file.

The various options are defined as:

-f
Selects *file mode* operation. The input stream is interpreted as a list of the names of the files to be processed instead of the data itself.

-r
Selects a *raw mode* of operation. This option is used mainly to determine if two versions of an executable file are *exactly* the same. This switch causes *crc* to include the compiler-generated time stamps in the *coff* file image when computing the *crc*.

-c
Changes the output to include the byte count of each file processed.

-d
Adds the time of the file's modification to the output.

-l
Computes the *crc* in decimal for *each line of the input file* instead of the whole file itself. *Use of this option overrides all others.*

Note that the first four options can be used in any combination. There are three general forms of output. The first form is produced without the **-c** option:

$nnnn for *filename (time stamp)*

where *$nnnn* is the 16-bit checksum in hexadecimal representation; and *time stamp* is the time of the file's modification (displayed if the **-d** option is selected). The fields are separated by space (20h) characters.

The second output form is generated when the **-c** option is selected:

$nnnn length time stamp filename

where *length* is the true size of the file, regardless of whether *raw mode* (**-r**) is selected; and *time stamp* is the time of the file's modification (displayed if the **-d** option is selected). All fields of this second form are delimited by tab (\t) characters.

The third form of output is produced by the *line mode* option (**-l**). It replaces each line of input with its corresponding *crc* in the form *nnnn*.

DIAGNOSTICS

crc: bad option letter. an invalid option letter was specified.

crc: argument count. at least one file name (or '-') must be provided.

crc: can't open file for reading. file cannot be opened for some reason.

crc: can't read file. input file cannot be read for some reason.

EXAMPLES

Suppose a **touch *; ls -log** command produces the following directory listing:

```
-rwxrwxrwx  1   23 Apr  8 12:39  apple
-rwxrwxrwx  1 8307 Apr  8 12:39  peaches
-rwxrwxrwx  1 1280 Apr  8 12:39  pears
-rwxrwxrwx  1  771 Apr  8 12:39  plums
```

Note that **ls | crc -fdc -** *is equivalent to* **crc -dc ***, and both would produce output similar to:

```
$8AC3    23   Apr  8 12:39:51 1986   apple
$FD06  8307   Apr  8 12:39:51 1986    peaches
$C3B0  1280   Apr  8 12:39:51 1986    pears
$02D2   771   Apr  8 12:39:51 1986     plums
```

A means of generating checksums for an entire directory hierarchy is:

find *root_path* **-type f -print | crc -fcd -**

Use of the '-type f' option on *find* is recommended because *crc* will generate the *crc* for the directory files themselves if presented with their names.

You can extract just the *crc* and length from a stream of *crc's* by using the *cut* command. When appended to the above command,

find *args* **-type f -print | crc -fcd - | cut -f1,2**

produces an output of two columns: the *crc* and the file's length.

FILES

/usr/bin/crc

NAME

crontab – user crontab file

SYNOPSIS

crontab *[file]*
crontab –r
crontab **–l**

DESCRIPTION

crontab copies the specified file, or standard input if no file is specified, into a directory that holds all users' **crontabs**. The **–r** option removes a user's **crontab** from the **crontab** directory. **crontab –l** will list the **crontab** file for the invoking user.

Users are permitted to use *crontab* if their names appear in the file **/usr/lib/cron/cron.allow.** If that file does not exist, the file **/usr/lib/cron/cron.deny** is checked to determine if the user should be denied access to **crontab**. If neither file exists, only **root** is allowed to submit a job. If **cron.allow** does not exist and **cron.deny** exists but is empty, global usage is permitted. The allow/deny files consist of one user name per line.

A **crontab** file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0–59),
hour (0–23),
day of the month (1–31),
month of the year (1–12),
day of the week (0–6 with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to. For example, 0 0 1,15 * 1 would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example, 0 0 * * 1 runs a command only on Mondays).

The sixth field of a line in a **crontab** file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is translated to a newline character. Only the first line (up to a % or end of line) of the command field is executed by the shell. The other lines are made available to the command as standard input.

The shell is invoked from your $HOME directory with an **arg0** of *sh*. Users who desire to have their **.profile** executed must explicitly do so in the **crontab** file. *cron* supplies a default environment for every shell, defining: **HOME, LOGNAME, SHELL(=/bin/sh),** and **PATH(=:/bin:/usr/bin:/usr/lbin)**.

If you do not redirect the standard output and standard error of your commands, any generated output or errors will be mailed to you.

FILES

/usr/lib/cron	main cron directory
/usr/spool/cron/crontabs	spool area
/usr/lib/cron/log	accounting information
/usr/lib/cron/cron.allow	list of allowed users
/usr/lib/cron/cron.deny	list of denied users

SEE ALSO

sh(1).
cron(1M) in the *System Administrator's Reference Manual*.

WARNINGS

If you inadvertently enter the *crontab* command with no argument(s), do not attempt to get out with a `ctrl-d`. This will cause all entries in your **crontab** file to be removed. Instead, exit with a `DEL`.

NAME

crypt – encode/decode

SYNOPSIS

crypt [*password*]
crypt [–k]

DESCRIPTION

crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no argument is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. If the **–k** option is used, *crypt* will use the key assigned to the environment variable CRYPTKEY. *crypt* encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

Files encrypted by *crypt* are compatible with those treated by the editors *ed*(1), *edit*(1), *ex*(1), and *vi*(1) in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; "sneak paths" by which keys or clear text can become visible must be minimized.

crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e., to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lowercase letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

If the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. The choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty **for typed key**

SEE ALSO

ed(1), edit(1), ex(1), makekey(1), ps(1), stty(1), vi(1).

WARNING

This command is provided with the Security Administration Utilities, which is only available in the United States. If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

BUGS

If output is piped to *nroff* and the encryption key is *not* given on the command line, *crypt* can leave terminal modes in a strange state (see *stty*(1)).

NAME

csplit – context split

SYNOPSIS

csplit [**–s**] [**–k**] [**–f** *prefix*] *file arg1 [. . . argn]*

DESCRIPTION

csplit reads *file* and separates it into n+1 sections, defined by the arguments *arg1. . . argn*. By default, the sections are placed in xx00 . . . xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00:
From the start of *file* up to (but not including) the line referenced by *arg1*.

01:
From the line referenced by *arg1* up to the line referenced by *arg2*.

.
.
.

n+1:
From the line referenced by *argn* to the end of *file*.

If the *file* argument is a **–**, standard input is used.

The options to *csplit* are:

–s
csplit normally prints the character counts for each file created. If the **–s** option is present, *csplit* suppresses the printing of all character counts.

–k
csplit normally removes created files if an error occurs. If the **–k** option is present, *csplit* leaves previously created files intact.

–f *prefix*
If the **–f** option is used, the created files are named *prefix***00 . . .** *prefixn*. The default is **xx00 . . . xx***n*.

The arguments (*arg1 . . . argn*) to *csplit* can be a combination of the following:

/rexp/
A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*.

The current line becomes the line containing *rexp*. This argument may be followed by an optional + or – some number of lines (e.g., **/Page/–5**).

%rexp%
This argument is the same as */rexp/*, except that no file is created for the section.

lnno
A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{num}
Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded new-lines. *csplit* does not affect the original file; it is the users responsibility to remove it.

EXAMPLES

csplit –f cobol file '/procedure division/' /par5./ /par16./

This example creates four files, **cobol00 . . . cobol03**. After editing the "split" files, they can be recombined as follows:

cat cobol0[0–3] > file

Note that this example overwrites the original file.

csplit –k file 100 {99}

This example would split the file at every 100 lines, up to 10,000 lines. The **–k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

csplit –k prog.c '%main(%' '/^}/+1' {20}

Assuming that **prog.c** follows the normal **C** coding convention of ending routines with a } at the beginning of the line, this example will create a file containing each separate **C** routine (up to 21) in **prog.c**.

SEE ALSO

ed(1), sh(1), split(1).
regexp(5) in the *Programmer's Reference Manual.*

DIAGNOSTICS

Self-explanatory except for:

arg – out of range

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

ctags – maintain a tags file for a C program

SYNOPSIS

ctags [**–a**] [**–u**] [**–w**] [**–x**] *name* ...

DESCRIPTION

ctags makes a tags file for *ex* (1) and *vi* (1) from the specified sources.

A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the tags file, *ex* can quickly find these function definitions.

Options

–a
appends the output to the tags file instead of rewriting it.

–u
causes the specified files to be *updated* in tags, that is, all references to them are replaced by new values. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

–w
suppresses warning diagnostics.

–x
produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output.

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions.

The tag *main* is treated specially in C programs. The tag formed is created by prefixing M to the name of the file, with a trailing **.c** removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

EXAMPLE

ctags *.c *.h

puts the tags from all the **.c** and **.h** files into the tags file **tags**.

FILES

/usr/bin/ctags	
tags	output tags file

SEE ALSO

ex(1), vi(1).

BUGS

Not all warning diagnostics are suppressed by **–w**.

If *ctags* is interrupted while executing under the **–u** option, a temporary file named **OTAGS** is left in the current directory.

NAME

ctrace – C program debugger

SYNOPSIS

ctrace *[options] [file]*

DESCRIPTION

The *ctrace* command allows you to follow the execution of a C program, statement-by-statement. The effect is similar to executing a shell procedure with the **–x** option. *ctrace* reads the C program in **file** (or from standard input if you do not specify **file**), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of *ctrace* into a temporary file because the *cc*(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output so you can put it into a file for examination with an editor or the *bfs*(1) or *tail*(1) commands.

The options commonly used are:

–f *functions*
Trace only these *functions.*

–v *functions*
Trace all but these *functions.*

You may want to add to the default formats for printing variables. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. Char, short, and int variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation. You can request that variables be printed in additional formats, if appropriate, with the following options:

–o
Octal

–x
Hexadecimal

–u
Unsigned

–e
Floating point

The following options are used only in special circumstances:

–l *n*
Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

–s
Suppress redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.

–t *n*
Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The **DIAGNOSTICS** section explains when to use this option.

–P
Run the C preprocessor on the input before tracing it. You can also use the **–D**, **–I**, and **–U** *cpp*(1) options.

These options are used to tailor the run-time trace package when the traced program will run in a non-UNIX System environment:

–b
Use only basic functions in the trace code, i.e., those in *ctype*(3C), *printf*(3S), and *string*(3C). These are usually available even in cross-compilers for microprocessors. In particular, this option is needed when the traced program runs under an operating system that does not have *signal*(2), *fflush*(3S), *longjmp*(3C), or *setjmp*(3C).

–p *string*
Change the trace print function from the default of 'printf('. For example, 'fprintf(stderr,' would send the trace to the standard error output.

–r *f*
Use file *f* in place of the *runtime.c* trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the **–p** option).

EXAMPLE

If the file *lc.c* contains this C program:

```
 1 #include <stdio.h>
 2 main()    /* count lines in input */
 3 {
 4     int c, nl;
 5
 6     nl = 0;
 7     while ((c = getchar()) != EOF)
 8           if (c = '\n')
 9                 ++nl;
10     printf("%d\n", nl);
11 }
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
(CTRL-d)
```

the program will be compiled and executed. The output of the program will be the number **2**, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke *ctrace* with these commands:

```
ctrace lc.c >temp.c
cc temp.c
a.out
```

the output will be:

```
 2 main()
 6     nl = 0;
       /* nl == 0 */
 7     while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```
        /* c == 49 or '1' */
8               if (c = '\n')
                /* c == 10 or '\n' */
9                       ++nl;
                        /* nl == 1 */
7       while ((c = getchar()) != EOF)
        /* c == 10 or '\n' */
8               if (c = '\n')
                /* c == 10 or '\n' */
9                       ++nl;
                        /* nl == 2 */
7       while ((c = getchar()) != EOF)
```

If you now enter an end of file character (**CTRL-d**), the final output will be:

```
        /* c == -1 */
10      printf("%d\n", nl);
        /* nl == 2 */2
         return
```

Note that the program output printed at the end of the trace line for the **nl** variable. Also note the **return** comment added by *ctrace* at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable **c** is assigned the value '1' in line 7, but in line 8 it has the value '\n'. Once your attention is drawn to this **if** statement, you will probably realize that you used the assignment operator (=) in place of the equality operator (==). You can easily miss this error during code reading.

EXECUTION-TIME TRACE CONTROL

The default operation for *ctrace* is to trace the entire program file, unless you use the **–f** or **–v** options to trace specific functions. This does not give you statement-by-statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding *ctroff*() and *ctron*() function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with *if* statements, and you can even conditionally include this code because *ctrace* defines the CTRACE preprocessor variable. For example:

```
#ifdef CTRACE
        if (c == '!' && i > 1000)
                ctron();
#endif
```

You can also call these functions from *sdb*(1) if you compile with the **–g** option. For example, to trace all but lines 7 to 10 in the main function, enter:

```
sdb a.out
main:7b ctroff()
main:11b ctron()
r
```

You can also turn the trace off and on by setting static variable tr_ct_ to 0 and 1, respectively. This is useful if you are using a debugger that cannot call these functions directly.

DIAGNOSTICS

This section contains diagnostic messages from both *ctrace* and *cc*(1), since the traced code often gets some *cc* warning messages. You can get *cc* error messages in some rare cases, all of which can be avoided.

ctrace Diagnostics

warning: some variables are not traced in this statement

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the **–t** option to increase this number.

warning: statement too long to trace

This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

cannot handle preprocessor code, use –P option

This is usually caused by **`#ifdef/#endif`** preprocessor statements in the middle of a C statement, or by a semicolon at the end of a **`#define`** preprocessor statement.

'if ... else if' sequence too long

Split the sequence by removing an **else** from the middle.

possible syntax error, try –P option

Use the **–P** option to preprocess the *ctrace* input, along with any appropriate **–D**, **–I**, and **–U** preprocessor options. If you still get the error message, check the WARNINGS section.

Cc Diagnostics

warning: illegal combination of pointer and integer

warning: statement not reached

warning: sizeof returns 0
Ignore these messages.

compiler takes size of function
See the *ctrace* "possible syntax error" message above.

yacc stack overflow
See the *ctrace* "'if ... else if' sequence too long" message above.

out of tree space; simplify expression
Use the **-t** option to reduce the number of traced variables per statement from the default of 10. Ignore the **`ctrace: too many variables to trace`** warnings you will now get.

redeclaration of signal
Either correct this declaration of *signal*(2), or remove it and **`#include <signal.h>`**.

SEE ALSO

signal(2), ctype(3C), fclose(3S), printf(3S), setjmp(3C), string(3C).
bfs(1), tail(1) in the *User's Reference Manual*.

WARNINGS

You will get a *ctrace* syntax error if you omit the semicolon at the end of the last element declaration in a structure or union, just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. Just use a different name.

ctrace assumes that BADMAG is a preprocessor macro, and that EOF and **`NULL`** are #defined constants. Declaring any of these to be variables, e.g., **`int`** EOF;, will cause a syntax error.

BUGS

ctrace does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all the components of an aggregate when an assignment is made to the entire aggregate. *ctrace* may choose to print the address of an aggregate or use the wrong format (e.g., 3.149050e-311 for a structure with two integer members) when printing the value of an aggregate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

/usr/lib/ctrace/runtime.c run-time trace package

NAME

cut – cut out selected fields of each line of a file

SYNOPSIS

cut **–c***list* [*file ...*]
cut **–f***list* [**–d***char*] [**–s**] [*file ...*]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (**–c** option) or the length can vary from line to line and be marked with a field delimiter character like *tab* (**–f** option). *cut* can be used as a filter; if no files are given, the standard input is used. In addition, a file name of "–" explicitly refers to standard input.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional – to indicate ranges (e.g., **1,4,7; 1–3,8; –5,10** (short for **1–5,10**); or **3–** (short for third through last field)).

–c*list* The *list* following **–c** (no space) specifies character positions (e.g., **–c1–72** would pass the first 72 characters of each line).

–f*list* The *list* following **–f** is a list of fields assumed to be separated in the file by a delimiter character (see **–d**); e.g., **–f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless **–s** is specified.

–d*char* The character following **–d** is the field delimiter (**–f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

–s Suppresses lines with no delimiter characters in case of **–f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the **–c** or **–f** option must be specified.

Use *grep*(1) to make horizontal "cuts" (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut –d: –f1,5 /etc/passwd	mapping of user IDs to names
name=`who am i | cut –f1 –d" "`	to set **name** to current login name.

DIAGNOSTICS

ERROR: line too long
A line can have no more than 1023 characters or fields, or there is no newline character.

ERROR: bad list for c/f option
Missing **–c** or **–f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

ERROR: no fields
The *list* is empty.

ERROR: no delimeter
Missing *char* on **–d** option.

ERROR: cannot handle multiple adjacent backspaces
Adjacent backspaces cannot be processed correctly.

WARNING: cannot open <filename>
Either *filename* cannot be read or does not exist. If multiple file names are present, processing continues.

SEE ALSO

grep(1), paste(1).

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref *[options] files*

DESCRIPTION

The *cxref* command analyzes a collection of C files and attempts to build a cross-reference table. *cxref* uses a special version of *cpp* to include **#defined** information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or, with the **–c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **–D, –I** and **–U** options (which are interpreted just as they are by *cc*(1) and *cpp*(1)), the following *options* are interpreted by *cxref*:

–c Print a combined cross-reference of all input files.

–w<*num*>
Width option which formats output no wider than <num> (decimal) columns. This option will default to 80 if <num> is not specified or is less than 51.

–o file Direct output to *file.*

–s Operate silently; do not print input file names.

–t Format listing for 80-column width.

FILES

LLIBDIR	usually **/usr/lib**
LLIBDIR/xcpp	special version of the C preprocessor.

SEE ALSO

cc(1), cpp(1).

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

BUGS

cxref considers a formal argument in a `#define` macro definition to be a declaration of that symbol. For example, a program that `#includes` **ctype.h**, will contain many declarations of the variable **c**.

NAME

date – print and set the date

SYNOPSIS

date [+ *format*]
date [*mmddhhmm*[[*yy*] | [*ccyy*]]]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set (only by superuser). 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; *cc* is the century minus one and is optional; *yy* is the last 2 digits of the year number and is optional. The following example sets the date to Oct 8, 12:45 AM:

date 10080045

The current year is the default if no year is mentioned. The system operates in GMT. *date* takes care of the conversion to and from local standard and daylight time. Only the superuser may change the date.

If the argument begins with +, the output of *date* is under the control of the user. All output fields are of fixed size (zero padded if necessary). Each Field Descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a newline character. If the argument contains embedded blanks it must be quoted (see the **EXAMPLE** section).

Specifications of native language translations of month and weekday names are supported. The language used depends on the value of the environment variable **LANGUAGE** (see *environ*(5)). The month and weekday names used for a language are taken from strings in the file for that language in the **/lib/cftime** directory (see *cftime*(4)).

After successfully setting the date and time, *date* will display the new date according to the format defined in the environment variable **CFTIME** (see *environ*(5)).

Field Descriptors (must be preceded by a %):

a	abbreviated weekday name
A	full weekday name
b	abbreviated month name
B	full month name
d	day of month; 01 to 31
D	date as mm/dd/yy
e	day of month; 1 to 31 (single digits are preceded by a blank)
h	abbreviated month name (alias for %b)
H	hour; 00 to 23
I	hour; 01 to 12
j	day of year; 001 to 366
m	month of year; 01 to 12
M	minute; 00 to 59
n	insert a newline character
p	string containing ante-meridiem or post-meridiem indicator (by default, AM or PM)
r	time as *hh:mm:ss pp* where *pp* is the ante-meridiem or post-meridiem indicator (by default, AM or PM)
R	time as hh:mm
S	second; 00 to 59
t	insert a tab character
T	time as *hh:mm:ss*
U	week number of year (Sunday as the first day of the week); 01 to 52
w	day of week; Sunday = 0
W	week number of year (Monday as the first day of the week); 01 to 52
x	Country-specific date format
X	Country-specific time format
y	year within century; 00 to 99
Y	year as *ccyy* (4 digits)
Z	timezone name

EXAMPLE

date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'

Generates as output:

```
DATE: 08/01/76
TIME: 14:45:05
```

DIAGNOSTICS

No permission — if you are not the superuser and you try to change the date

bad conversion — if the date set is syntactically incorrect

FILES

/dev/kmem

WARNING

Should you need to change the date while the system is running multi-user, use *sysadm*(1) *datetime*.

NOTE

Administrators should note the following: if you attempt to set the current date to one of the dates that the standard and alternate time zones change (e.g., the date that daylight time is starting or ending), and you attempt to set the time to a time in the interval between the end of standard time and the beginning of the alternate time (or the end of the alternate time and the beginning of standard time), the results are unpredictable.

SEE ALSO

sysadm(1)

cftime(4), environ(5) in the *System Administrator's Reference Manual.*

NAME

dc – desk calculator

SYNOPSIS

dc [*file*]

DESCRIPTION

dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. (See *bc*(1), a preprocessor for *dc* that provides infix notation and a C-like syntax that implements functions; *bc* also provides reasonable control structures for programs.) The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
The value of the number is pushed on the stack. A number is an unbroken string of the digits 0–9. It may be preceded by an underscore (_) to input a negative number. Numbers may contain decimal points.

+ – / * % ^
The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x*
The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x*
The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the l is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d
The top value on the stack is duplicated.

p
The top value on the stack is printed. The top value remains unchanged.

P
Interprets the top of the stack as an ASCII string, removes it, and prints it.

f
All values on the stack are printed.

q
Exits the program. If executing a string, the recursion level is popped by two.

Q
Exits the program. The top value on the stack is popped and the string execution level is popped by that value.

x
Treats the top element of the stack as a character string and executes it as a string of *dc* commands.

X
Replaces the number on the top of the stack with its scale factor.

[...]
Puts the bracketed ASCII string onto the top of the stack.

<*x* >*x* =*x*
The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation.

v
Replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, otherwise, the scale factor is ignored.

!
Interprets the rest of the line as a command to the shell.

c
All values on the stack are popped.

i
The top value on the stack is popped and used as the number radix for further input. I pushes the input base on the top of the stack.

o
The top value on the stack is popped and used as the number radix for further output.

O
Pushes the output base on the top of the stack.

k
The top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z
The stack level is pushed onto the stack.

Z
Replaces the number on the top of the stack with its length.

?
A line of input is taken from the input source (usually the terminal) and executed.

; :
Used by *bc*(1) for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1).

DIAGNOSTICS

x is unimplemented
where *x* is an octal number.

stack empty
for not enough elements on the stack to do what was asked.

Out of space
when the free list is exhausted (too many digits).

Out of headers
for too many numbers being kept around.

Out of pushdown
for too many items on the stack.

Nesting Depth
for too many levels of nested execution.

NAME

dcpy – copy removeable media

SYNOPSIS

dcpy *from to*

DESCRIPTION

dcpy provides a method for non-root users to make copies of removeable media file systems. To actually copy the file systems, it calls the *dcopy* utility.

There are no options for *dcpy*. *from* and *to* are the names of removable media filesystems, and must be specified.

The *from* filesystem must be present and readable.

The *to* filesystem must be present and writeable.

The blocksize of the *to* filesystem will be made the same as the blocksize of the *from* filesystem. *dcpy* itself uses 1Kb blocks.

FILES

/etc/filesys	permissions file
/etc/dcopy	system utility program

SEE ALSO

dcopy(1M) in the *System Administrators Reference Manual.*

DIAGNOSTICS

dcpy will complain, `No read permission on input file.` if the input file (the *from* file) does not exist or is not readable.

dcpy will complain, `No write permission on output file.` if the output device (the *to* file) does not exist or is not readable.

dcpy will print `Failed to exec dcopy`" if, for some reason, it was unable to execute **/etc/dcopy.**

Additional error messages may be printed by the *dcopy* program itself.

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [–r*SID*] [–s] [–n] [–g*list*] [–m[*mrlist*]] [–y[*comment*]] [–p] *files*

DESCRIPTION

delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

delta makes a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see **WARNINGS**); each line of the standard input is taken to be the name of an SCCS file to be processed.

delta may issue prompts on the standard output depending upon certain keyletters specified and flags (see *admin*(1)) that may be present in the SCCS file (see **–m** and **–y** keyletters below).

Keyletter arguments apply independently to each named file.

–r*SID*

Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (**get –e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **–r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(1)). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–s

Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

–n

Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

–g*list*
a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

–m*[mrlist]*
If the SCCS file has the **v** flag set (see *admin*(1)) then a **MR** number *must* be supplied as the reason for creating the new delta.

If **–m** is not used and the standard input is a terminal, the prompt **MRs?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The **MRs?** prompt always precedes the **comments?** prompt (see –y keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the **v** flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) that will validate the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *delta* terminates. (It is assumed that the MR numbers were not all valid.)

–y*[comment]*
Arbitrary text used to describe the reason for making the delta. A **NULL** string is considered a valid *comment*.

If **–y** is not specified and the standard input is a terminal, the prompt **comments?** is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the comment text.

–p
Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the **g-file**.

WARNINGS

Lines beginning with an **SOH** ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS (see *sccsfile*(4) (5)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get*/*delta* sequences should be used.

If the standard input (–) is specified on the *delta* command line, the **–m** (if necessary) and **–y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

SEE ALSO

admin(1), cdc(1), get(1), prs(1), rmdel(1), sccsfile(4).
bdiff(1), help(1) in the *User's Reference Manual*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

deroff – remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [**–mx**] [**–w**] [*files*]

DESCRIPTION

deroff reads each of the *files* in sequence and removes all *troff*(1) requests, macro calls, backslash constructs, *eqn*(1) constructs (between **.EQ** and **.EN** lines, and between delimiters), and *tbl*(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output. *deroff* follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The **–m** option may be followed by an **m**, **s**, or **l**. The **–mm** option causes the macros to be interpreted so that only running text is output (i.e., no text from macro lines). The **–ml** option forces the **–mm** option and also causes deletion of lists associated with the **mm** macros.

If the **–w** option is given, the output is a word list, one "word" per line, with all other characters deleted. Otherwise, the output follows the original, with the deletions mentioned above. In text, a "word" is any string that *contains* 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 contains 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."

SEE ALSO

eqn(1), nroff(1), tbl(1), troff(1) in the *DOCUMENTER'S WORKBENCH Software Release 2.0 Technical Discussion and Reference Manual.*

BUGS

deroff is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most such errors result in too much rather than too little output.

The **–ml** option does not handle nested lists correctly.

NAME

diff – differential file comparator

SYNOPSIS

diff [**–efbh**] *file1 file2*

DESCRIPTION

diff tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is **–**, the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory with the name *file2* (*file1*) is used. The normal output contains lines of these forms:

```
n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4
```

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging **a** for **d** and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs, where *n1* = *n2* or *n3* = *n4*, are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by <, then all the lines that are affected in the second file flagged by >.

The **–b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The **–e** option produces a script of *a*, *c*, and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The **–f** option produces a similar script, not useful with *ed*, in the opposite order. In connection with **–e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A "latest version" appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed – $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option **–h** does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options **–e** and **–f** are unavailable with **–h**.

FILES

/tmp/d?????
/usr/lib/diffh for –h

SEE ALSO

bdiff(1), cmp(1), comm(1), ed(1).

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

BUGS

Editing scripts produced under the **–e** or **–f** option are naive about creating lines consisting of a single period (.).

WARNINGS

Missing newline at end of file X
indicates that the last line of file X did not have a newline. If the lines are different, they will be flagged and output; although the output will seem to indicate they are the same.

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**–ex3**] *file1 file2 file3*

DESCRIPTION

diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f **:** *n1* **a**
Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.

f **:** *n1* **,** *n2* **c**
Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **–e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, i.e., the changes that normally would be flagged ==== and ====3. Option **–x** (**–3**) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*:

(cat script; echo '1,$p') | ed – file1

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1).

BUGS

Text lines that consist of a single . will defeat **–e**.
Files longer than 64Kb will not work.

NAME

diffmk – mark differences between files

SYNOPSIS

diffmk *name1 name2 name3*

DESCRIPTION

diffmk is a shell procedure that compares two versions of a file and creates a third file that includes "change mark" commands for *nroff* or *troff*(1). *Name1* and *name2* are the old and new versions of the file. *diffmk* generates *name3*, which contains the lines of *name2* plus inserted formatter "change mark" (**.mc**) requests. When *name3* is formatted, changed or inserted text is shown by | at the right margin of each line. The position of deleted text is shown by a single *.

diffmk can be used to produce listings of C (or other) programs with changes marked. A typical command line for such use is:

diffmk old.c new.c tmp; nroff macs tmp | pr

where the file *macs* contains:

```
.pl 1
.ll 77
.nf
.eo
.nc
```

The **.ll** request can be used to specify a different line length, depending on the nature of the program being printed. The **.eo** and **.nc** requests are probably needed only for C programs.

If the characters | and * are inappropriate, a copy of *diffmk* can be edited to change them.

SEE ALSO

diff(1), nroff(1), troff(1).

BUGS

Aesthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output, i.e., replacing **.sp** by **.sp 2** produces a "change mark" on the preceding or following line of output.

NAME

dircmp – directory comparison

SYNOPSIS

dircmp [**–d**] [**–s**] [**–w***n*] *dir1 dir2*

DESCRIPTION

dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the file names common to both directories have the same contents.

–d
Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

–s
Suppress messages about identical files.

–w*n*
Change the width of the output line to *n* characters. The default width is 72.

SEE ALSO

cmp(1), diff(1).

NAME

dis – object code disassembler

SYNOPSIS

dis [**–o**] [**–V**] [**–L**] [**–s**] [**–d** *sec*] [**–da** *sec*] [**–F** *function*] [**–t** *sec*] [**–l** *string*] *file* ...

DESCRIPTION

The *dis* command produces an assembly language listing of *file*, which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

The following *options* are interpreted by the disassembler and may be specified in any order.

–o
Print numbers in octal. The default is hexadecimal.

–V
Print, on standard error, the version number of the disassembler being executed.

–L
Lookup source labels in the symbol table for subsequent printing. This option works only if the file was compiled with additional debugging information (e.g., the **–g** option of *cc*(1)).

–s
Perform symbolic disassembly- i.e., specify source symbol names for operands where possible. Symbolic disassembly output will appear on the line following the instruction. For maximal symbolic disassembly to be performed, the file must be compiled with additional debugging information (e.g., the **–g** option of *cc*(1)). Symbol names will be printed using C syntax.

–d *sec*
Disassemble the named section as data, printing the offset of the data from the beginning of the section.

–da *sec*
Disassemble the named section as data, printing the actual address of the data.

–F *function*
Disassemble only the named function in each object file specified on the command line. The **–F** option may be specified multiple times on the command line.

–t *sec*
Disassemble the named section as text.

–l *string*
Disassemble the library file specified by *string*. For example, one would issue the command **dis –l x –l z** to disassemble **libx.a** and **libz.a**. All libraries are assumed to be in **LIBDIR**.

If the **–d, –da** or **–t** options are specified, only those named sections from each user-supplied file name will be disassembled. Otherwise, all sections containing text will be disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as **[5]**, represents that the break-pointable line number starts with the following instruction. These line numbers will be printed only if the file was compiled with additional debugging information (e.g., the **–g** option of *cc*(1)). An expression such as **<40>** in the operand field or in the symbolic disassembly, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A function name will appear in the first column, followed by **()**.

FILES

LIBDIR usually /lib.

SEE ALSO

as(1), cc(1), ld(1), a.out(4).

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

NAME

dnp – patch program with `NULL` pointer dereference bug

SYNOPSIS

dnp *filename*

DESCRIPTION

On the SYSTEM V/88 BOS, the kernel is set up so that any program that attempts to dereference a pointer at a `NULL` (or very low) address will dump core. This is because the SYSTEM V/88 BOS system does not guarantee there will be zeroes at address 0. However, a workaround has been provided in the kernel. If the *timdat* field in the file header (filehdr) structure has a value of 1, the kernel will provide a block of zeroes at address 0 to allow the program to run until the source code can be fixed.

The *dnp* program changes the value of the *timdat* field in an executable file to 1, so that the utility may be run.

When the *dnp* utility is run, several diagnostic messages are printed: the pathname of the file, the *magic number* of the file to be patched (only files with magic number 0555 can be patched), and the old and new values of the *timdat* field.

The *file* command recognizes a file that has been patched in this way, indicating that it is *dnp kludged.*

EXAMPLES

dnp foo

FILES

/usr/lbin/dnp

DIAGNOSTICS

can't change non-0555 files!

When the file you are attempting to patch is not a current SYSTEM V/88 executable file with a *magic number* of 0555.

Error: fopen on *filename* failed

If the file you are attempting to patch, does not exist or cannot be read or written.

NOTES

The *dnp* utility should only be used to patch programs until the source code can be fixed.

Utilities may dump core for reasons other than **NULL** pointer problems; if the utility continues to dump core after being *dnp* patched, there is some other problem.

Please be sure to report to your representative any utilities in which you discover **NULL** pointer dereference problems, so that they may be fixed in code.

Library routines (in *libc.a* and other libraries) may dump core with a **NULL** pointer if they are passed **NULL** pointers; for example, a *strcmp*() where one of the arguments is a **NULL** pointer. This should be fixed in the code that calls *strcmp*().

NAME

dump – dump selected parts of an object file

SYNOPSIS

dump [*options*] *files*

DESCRIPTION

The *dump* command dumps selected parts of each of its object *file* arguments.

This command will accept both object files and archives of object files. It processes each file argument according to one or more of the following options:

–a
Dump the archive header of each member of each archive file argument.

–g
Dump the global symbols in the symbol table of an archive.

–f
Dump each file header.

–o
Dump each optional header.

–h
Dump section headers.

–s
Dump section contents.

–r
Dump relocation information.

–l
Dump line number information.

–t
Dump symbol table entries.

–z *name*
Dump line number entries for the named function.

–c
Dump the string table.

–L
Interpret and print the contents of the *.lib* sections.

The following *modifiers* are used in conjunction with the options listed above to modify their capabilities.

–d *number*
Dump the section number, *number*, or the range of sections starting at *number* and ending at the *number* specified by **+d**.

+d *number*
Dump sections in the range either beginning with first section or beginning with section specified by **–d**.

–n *name*
Dump information pertaining only to the named entity. This *modifier* applies to **–h**, **–s**, **–r**, **–l**, and **–t**.

–p
Suppress printing of the headers.

–t *index*
Dump only the indexed symbol table entry. The **–t** used in conjunction with **+t**, specifies a range of symbol table entries.

+t *index*
Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **–t** option.

–u
Underline the name of the file for emphasis.

–v
Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of 0X02). This *modifier* can be used with all the above options except **–s** and **–o** options of *dump*.

–z *name,number*
Dump line number entry or range of line numbers starting at *number* for the named function.

+z *number*
Dump line numbers starting at either function *name* or *number* specified by **–z**, up to *number* specified by **+z**.

Blanks separating an *option* and its *modifier* are optional. The comma separating the name from the number modifying the **–z** option may be replaced by a blank.

The *dump* command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal or decimal representation as appropriate.

SEE ALSO

a.out(4), ar(4).

NAME

echo – echo arguments

SYNOPSIS

echo *[arg] ...*

DESCRIPTION

echo writes its arguments separated by blanks and terminated by a newline on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b
backspace
\c
print line without newline
\f
form-feed
\n
newline
\r
carriage return
\t tab
\v
vertical tab
****
backslash
\0*n*
where *n* is the 8-bit character whose ASCII code is the 1-, 2-, or 3-digit octal number representing that character.

echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

CAVEATS

When representing an 8-bit character by using the escape convention **\0***n*, the *n* must **always** be preceded by the digit zero (0).

For example, typing: **echo ^WARNING:\07^** will print the phrase **"WARNING:"** and sound the "bell" on your terminal. The use of single (or double) quotes (or two backslashes) is required to protect the "\" that precedes the "07".

For the octal equivalents of each character, see *ascii*(5), in the *Programmer's Reference Manual*.

NAME

ed, red – text editor

SYNOPSIS

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

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

DESCRIPTION

ed is the standard text editor. If the *file* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited.

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

–p
Allows the user to specify a prompt string.

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

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

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

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

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

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

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

NOTE: When you are entering text into the file, this format is not in effect; instead, because of being in **stty –tabs** or **stty tab3** mode, tabs are expanded to every eighth column.

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

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

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

The following *one-character* RE s match a *single* character:

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

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

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

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([]) (see 1.4 below).

c. $ (dollar sign), which is special at the *end* of an entire RE (see 3.2).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the *g* command, below.)

1.3 A period (.) is a one-character RE that matches any character except newline.

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character *except* newline and the remaining characters in the string. The ^ has this special meaning *only* if it occurs first in the string.

The minus (–) may be used to indicate a range of consecutive ASCII characters; for example, **[0–9]** is equivalent to **[0123456789]**. The – loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); e.g., **[]a–f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

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

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

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

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both):

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

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

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The NULL RE (e.g., //) is equivalent to the last RE encountered. See also the last paragraph before the **FILES** section.

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

1. The character . addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. '*x* addresses the line marked with the mark name character *x*, which must be an ASCII lowercase letter (a-z). Lines are marked with the *k* command described below.

5. A RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before the **FILES** section.

6. A RE enclosed in question marks (?) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before the **FILES**.

7. An address followed by a plus sign (+) or a minus sign (–) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. The plus sign may be omitted.

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

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

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

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

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

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

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

(.)a
<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the "appended" text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

C

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

(.,.)d

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

e *file*

The *e*dit command causes the entire contents of the buffer to be deleted, then the named file to be read in; . is set to the last line of the buffer. If no file name is given, the currently remembered file name, if any, is used (see the *f* command). The number of characters read is typed; *file* is remembered for possible use as a default file name in subsequent *e*, *r*, and *w* commands. If *file* is replaced by **!**, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. Such a *shell* command is *not* remembered as the current file name. See the **DIAGNOSTICS** section.

E *file*

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

f *file*

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

(1,$)g/*RE*/*command list*

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

(1,$)G/*RE*/

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

h

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

H

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

(.)i
<text>
.

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

(.,.+1)j

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

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be an ASCII lowercase letter (**a-z**). The address *'x* then addresses this line; **.** is unchanged.

(.,.)l

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

(.,.)m*a*

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

(.,.)n

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

(.,.)p

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

P

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

q

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

Q

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

(**$**)**r** *file*

The *r*ead command reads in the given file after the addressed line. If no file name is given, the currently remembered file name, if any, is used (see *e* and *f* commands). The currently remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. Address 0 is legal for *r* and causes the file to be read at the beginning of the buffer.

If the read is successful, the number of characters read is typed; **.** is set to the last line read in. If *file* is replaced by **!**, the rest of the line is taken to be a *shell* (*sh*(1)) command whose output is to be read. For example, "$r !ls" appends current directory to the end of the file being edited. Such a shell command is *not* remembered as the current file name.

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

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

If a number **n** appears after the command, only the **n** th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or newline may be used instead of / to delimit the RE and the *replacement*; **.** is left at the last line on which a substitution occurred. See also the last paragraph before **FILES**.

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

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

(.,.)t*a*

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

u

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

(1,$)v/*RE/command list*

This command is the same as the global command *g* except that the *command list* is executed with . initially set to every line that does *not* match the RE.

(1,$)V/*RE/*

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do *not* match the RE.

(1,$)w *file*

The *w*rite command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your *umask* setting (see *umask*(1)) dictates otherwise. The currently remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked.

If no file name is given, the currently remembered file name, if any, is used (see *e* and *f* commands); **.** is unchanged. If the command is successful, the number of characters written is typed. If *file* is replaced by **!**, the rest of the line is taken to be a shell (*sh*(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current file name.

X

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

($)=

The line number of the addressed line is typed; **.** is unchanged by this command.

!*shell command*

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

(.+1)<newline>

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

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

Some size limitations: 512 characters in a line, 256 characters in a global command list, and 64 characters in the pathname of a file (counting slashes). The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

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

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

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a newline, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

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

FILES

$ TMPDIR	if this environmental variable is not **NULL**, its value is used in place of
/usr/tmp	as the directory name for the temporary work file.
/usr/tmp	if **/usr/tmp** exists, it is used as the directory name for the temporary work file.
/tmp	if the environmental variable *TMPDIR* does not exist or is **null**, and if **/usr/tmp** does not exist, then **/tmp** is used as the directory name for the temporary work file.
ed.hup	work is saved here if the terminal is hung up.

NOTES

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

SEE ALSO

edit(1), ex(1), grep(1), sed(1), sh(1), stty(1), umask(1), vi(1).
fspec(4), regexp(5) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

? for command errors.

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

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

WARNINGS

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

BUGS

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

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

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

If the editor input is coming from a command file (e.g., **ed file < ed-cmd-file**), the editor will exit at the first failure.

NAME

edit – text editor (variant of ex for casual users)

SYNOPSIS

edit [**–r**] [**–x**] [**–C**] *name*...

DESCRIPTION

edit is a variant of the text editor *ex* recommended for new or casual users who wish to use a command-oriented editor. It operates precisely as *ex* (1) with the following options automatically set:

novice	ON
report	ON
showmode	ON
magic	OFF

These options can be turned on or off via the *set* command in *ex* (1).

–r
Recover file after an editor or system crash.

–x
Encryption option; when used the file will be encrypted as it is being written and will require an encryption key to be read. *edit* makes an educated guess to determine if a file is encrypted or not. See *crypt* (1). Also, see the WARNING section at the end of this manual page.

–C
Encryption option; the same as **–x** except that *edit* assumes files are encrypted.

The following brief introduction should help you get started with *edit*. If you are using a CRT terminal, you may want to learn about the display editor *vi*.

To edit the contents of an existing file, you begin with the command *edit name* to the shell. *edit* makes a copy of the file that you can then edit, and tells you how many lines and characters are in the file. To create a new file, you also begin with the command *edit* with a file name: *edit name*; the editor will tell you it is a [New File].

The *edit* command prompt is the colon (:), which you should see after starting the editor. If you are editing an existing file, then you will have some lines in *edit 's* buffer (its name for the copy of the file you are editing). When you start editing, *edit* makes the last line of the file the current line. Most commands to *edit* use the current line if you do not tell them which line to use. Thus if you say **print** (which can be abbreviated *p*) and type carriage return (as you should after all *edit* commands), the current line will be printed. If you *delete* (*d*) the current line, *edit* will print the new current line, which is usually the next line in the file. If you *delete* the last line, then the new last line becomes the current one.

If you start with an empty file or wish to add some new lines, then the *append* (*a*) command can be used. After you execute this command (typing a carriage return after the word *append*), *edit* will read lines from your terminal until you type a line consisting of just a dot (.); it places these lines after the current line. The last line you type then becomes the current line. The command *insert* (*i*) is like *append*, but places the lines you type before, rather than after, the current line.

edit numbers the lines in the buffer, with the first line having number 1. If you execute the command **1**, then *edit* will type the first line of the buffer. If you then execute the command **d**, *edit* will delete the first line, line 2 will become line 1, and *edit* will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the *substitute* (*s*) command: **s**/*old*/*new*/ where *old* is the string of characters you want to replace and *new* is the string of characters you want to replace *old* with.

The command *file* (*f*) will tell you how many lines there are in the buffer you are editing and will say [Modified] if you have changed the buffer. After modifying a file, you can save the contents of the file by executing a *write* (*w*) command. You can leave the editor by issuing a *quit* (*q*) command. If you run *edit* on a file, but do not change it, it is not necessary (but does no harm) to *write* the file back. If you try to *quit* from *edit* after modifying the buffer without writing it out, you will receive the message No write since last change (:quit! overrides), and *edit* will wait for another command. If you do not want to write the buffer out, issue the *quit* command followed by an exclamation point (**q!**). The buffer is then irretrievably discarded and you return to the shell.

By using the *d* and *a* commands and giving line numbers to see lines in the file, you can make any changes you want. You should learn at least a few more things, however, if you will use *edit* more than a few times.

The *change* (*c*) command changes the current line to a sequence of lines you supply (as in *append*, you type lines up to a line consisting of only a dot (.). You can tell *change* to change more than one line by giving the line numbers of the lines you want to change, i.e., **3,5c**. You can print lines this way too: **1,23p** prints the first 23 lines of the file.

The *undo* (*u*) command reverses the effect of the last command you executed that changed the buffer. Thus if you execute a **substitute** command that does not do what you want, type **u** and the old contents of the line will be restored. You can also *undo* an *undo* command. *edit* will give you a warning message when a command affects more than one line of the buffer. Note that commands such as *write* and *quit* cannot be undone.

To look at the next line in the buffer, type carriage return. To look at a number of lines, type `CTRL-d` (while holding down the control key, press **d**) rather than carriage return. This will show you a half-screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at nearby text by executing the z command. The current line will appear in the middle of the text displayed, and the last line displayed will become the current line; you can get back to the line where you were before you executed the **z** command by typing ´´. The z command has other options: z– prints a screen of text (or 24 lines) ending where you are; z+ prints the next screenful. If you want less than a screenful of lines, type **z.11** to display five lines before and five lines after the current line. (Typing z.*n*, when *n* is an odd number, displays a total of *n* lines, centered about the current line; when *n* is an even number, it displays *n*–1 lines, so that the lines displayed are centered around the current line.) You can give counts after other commands; for example, you can delete 5 lines starting with the current line with the command **d5**.

To find things in the file, you can use line numbers if you happen to know them; since the line numbers change when you insert and delete lines this is somewhat unreliable. You can search backwards and forwards in the file for strings by giving commands of the form */text/* to search forward for *text* or *?text?* to search backward for *text* . If a search reaches the end of the file without finding *text*, it wraps around and continues to search back to the line where you are. A useful feature here is a search of the form */^text/* which searches for *text* at the beginning of a line. Similarly */text$/* searches for *text* at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has the symbolic name dot (.); this is most useful in a range of lines as in **.,$p** which prints the current line plus the rest of the lines in the file. To move to the last line in the file, you can refer to it by its symbolic name $. Thus, the command **$d** deletes the last line in the file, no matter what the current line is. Arithmetic with line references is also possible. Thus the line **$–5** is the fifth before the last and **.+20** is 20 lines after the current line.

You can find out the current line by typing **.=** . This is useful if you want to move or copy a section of text within a file or between files. Find the first and last line numbers you wish to copy or move. To move lines 10 through 20, type **10,20d a** to delete these lines from the file and place them in a buffer named **a**. *edit* has 26 such buffers named **a** through **z**. To put the contents of buffer **a** after the current line, type **put a**. If you want to move or copy these lines to another file, execute an *edit* (*e*) command after copying the lines; following the *e* command with the name of the other file you wish to edit, i.e., *edit chapter2*. To copy lines without deleting them, use *yank* (*y*) in place of *d*. If the text you wish to move or copy is all within one file, it is not necessary to use named buffers. For example, to move lines 10 through 20 to the end of the file, type **10,20m $**.

SEE ALSO

ed(1), ex(1), vi(1)

WARNING

The encryption options are provided as a separate package only to source product customers in the United States.

NAME

egrep – search a file for a pattern using full regular expressions

SYNOPSIS

egrep *[options] full regular expression [file ...]*

DESCRIPTION

egrep (*expression grep*) searches files for a pattern of characters and prints all lines that contain that pattern. *egrep* uses full regular expressions (expressions that have string values that use the full set of alphanumeric and special characters) to match the patterns. It uses a fast deterministic algorithm that sometimes needs exponential space.

egrep accepts full regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A full regular expression followed by + that matches one or more occurrences of the full regular expression.
2. A full regular expression followed by ? that matches 0 or 1 occurrences of the full regular expression.
3. Full regular expressions separated by | or by a newline that match strings that are matched by any of the expressions.
4. A full regular expression that may be enclosed in parentheses () for grouping.

Be careful using the characters $, *, [, ^, |, (,), and \ in *full regular expression*, because they are also meaningful to the shell. It is safest to enclose the entire *full regular expression* in single quotes '...'.

The order of precedence of operators is [], then *?+, then concatenation, then | and newline.

If no files are specified, *egrep* assumes standard input. Normally, each line found is copied to the standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

–b

Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).

–c
Print only a count of the lines that contain the pattern.

–i
Ignore upper-lowercase distinction during comparisons.

–l
Print the names of files with matching lines once, separated by newlines. It does not repeat the names of files when the pattern is found more than once.

–n
Precede each line by its line number in the file (first line is 1).

–v
Print all lines except those that contain the pattern.

–e *special_expression*
Search for a *special expression* (*full regular expression* that begins with a –).

–f *file*
Take the list of *full regular expressions* from *file*.

SEE ALSO

ed(1), fgrep(1), grep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep* command, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h.**

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 taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

The *disable* command deactivates the named *printers*, disabling them from printing requests taken by *lp*(1). 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. Use *lpstat*(1) to find the status of printers. 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.

–r *reason*
Assign a *reason* for the disabling of the printers. This *reason* applies to all printers mentioned up to the next **–r** option. This *reason* is reported by *lpstat*(1). If the **–r** option is not present, then a default reason will be used.

–W
Disable the specified printers when the print requests currently printing have finished. This option cannot be used with the **–c** option.

FILES

/usr/spool/lp/*

SEE ALSO

lp(1), lpstat(1).

NAME

env – set environment for command execution

SYNOPSIS

env [–] *[name=value] ... [command args]*

DESCRIPTION

env obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

SEE ALSO

sh(1)
exec(2), profile(4), environ(5) in the *Programmer's Reference Manual*.

NAME

ex – text editor

SYNOPSIS

ex [**–s**] [**–v**] [**–t** *tag*] [**–r** *file*] [**–L**] [**–R**] [**–x**] [**–C**] [**–c** *command*] *file* . . .

DESCRIPTION

ex is the root of a family of editors: *ex* and *vi*. *ex* is a superset of *ed* with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(1), which is a command which focuses on the display-editing portion of *ex*.

For ed Users

If you have used *ed* (1) you will find that, in addition to having all of the *ed* (1) commands available, *ex* has a number of additional features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi*. Generally, the *ex* editor uses far more of the capabilities of terminals than *ed* (1) does, and uses the terminal capability data base (see *terminfo* (4)) and the type of the terminal you are using from the environmental variable TERM to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its **visual** command (which can be abbreviated *vi*) and which is the central mode of editing when using *vi* (1).

ex contains a number of features for easily viewing the text of the file. The z command gives easy access to windows of text. Typing `CTRL-d` causes the editor to scroll down a half-window of text and is more useful for quickly stepping through a file than just typing return. Of course, the screen-oriented **visual** mode gives constant access to editing context.

ex gives you help when you make mistakes. The *undo* (*u*) command allows you to reverse any single change which goes astray. *ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files, unless you edited them, so that you do not accidentally overwrite a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the telephone, you can use the editor *recover* command (or **–r** *file* option) to retrieve your work. This will get you back to within a few lines of where you left off.

ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the *next* (*n*) command to deal with each in turn. The *next* command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, file names in the editor may be formed with full shell metasyntax. The metacharacter '%' is also available in forming file names and is replaced by the name of the current file.

The editor has a group of buffers whose names are the ASCII lowercase letters (**a-z**). You can place text in these named buffers where it is available to be inserted elsewhere in the file. The contents of these buffers remain available when you begin editing a new file using the *edit* (*e*) command.

There is a command **&** in *ex* which repeats the last *substitute* command. In addition, there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

It is possible to ignore the case of letters in searches and substitutions. *ex* also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word "edit" if your document also contains the word "editor."

ex has a set of options which you can set to tailor it to your liking. One option which is very useful is the *autoindent* option that allows the editor to supply leading white space to align text automatically. You can then use **CTRL-d** as a backtab and space or tab to move forward to align new code easily.

Miscellaneous useful features include an intelligent *join* (*j*) command that supplies white space between joined lines automatically, commands "<" and ">" which shift groups of lines, and the ability to filter portions of the buffer through commands such as *sort* (1).

Invocation Options

The following invocation options are interpreted by *ex* (previously documented options are discussed in the **NOTES** section at the end of this manual page):

-s
Suppress all interactive-user feedback. This is useful in processing editor scripts.

–v
Invoke *vi*.

–t *tag*
Edit the file containing the *tag* and position the editor at its definition.

–r *file*
Edit *file* after an editor or system crash. (Recovers the version of *file* that was in the buffer when the crash occurred.)

–L
List the names of all files saved as the result of an editor or system crash.

–R
Readonly mode; the **readonly** flag is set, preventing accidental overwriting of the file.

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

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

–c *command*
Begin editing by executing the specified editor *command* (usually a search or positioning command).

The *file* argument indicates one or more files to be edited.

ex States

Command
Normal and initial state. Input prompted for by **:**. Your line kill character cancels a partial command.

Insert
Entered by **a**, **i**, or **c**. Arbitrary text may be entered. Insert state normally is terminated by a line having only "." on it, or, abnormally, with an interrupt.

Visual
Entered by typing *vi*; terminated by typing Q or ^\ (CTRL-\).

ex Command Names and Abbreviations

abbrev	**ab**	**map**		**set**	**se**
append	**a**	**mark**	**ma**	**shell**	**sh**
args	**ar**	**move**	**m**	**source**	**so**
change	**c**	**next**	**n**	**substitute**	**s**
copy	**co**	**number**	**nu**	**unabbrev**	**unab**
delete	**d**	**preserve**	**pre**	**undo**	**u**
edit	**e**	**print**	**p**	**unmap**	**unm**
file	**f**	**put**	**pu**	**version**	**ve**
global	**g**	**quit**	**q**	**visual**	**vi**
insert	**i**	**read**	**r**	**write**	**w**
join	**j**	**recover**	**rec**	**xit**	**x**
list	**l**	**rewind**	**rew**	**yank**	**ya**

ex Commands

forced encryption	C	heuristic encryption	X
resubst	&	print next	CR
rshift	>	lshift	<
scroll	CTRL-d	window	z
shell escape	!		

ex Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	*?pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
–	previous	*'x*	marked with *x*
+n	*n* forward	''	previous context
%	1,$		

Initializing options

EXINIT	place **set**'s here in environment variable
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option *x*
set no*x*	disable option *x*
set *x*=*val*	give value *val* to option *x*
set	show changed options
set all	show all options
set *x***?**	show value of option *x*

Most useful options and their abbreviations

autoindent	**ai**	supply indent
autowrite	**aw**	write before changing files
directory		pathname of directory for temporary work files
exrc	**ex**	allow **vi/ex** to read the **.exrc** in the current directory. This option is set in the EXINIT shell variable or in the **.exrc** file in the $HOME directory.
ignorecase	**ic**	ignore case of letters in scanning
list		print ^I for tab, $ at end
magic		treat . [* special in patterns
modelines		first five lines and last five lines executed as *vi/ex* commands if they are of the form: **ex:***command***:** or **vi:***command***:**
number	**nu**	number lines
paragraphs	**para**	macro names that start paragraphs
redraw		simulate smart terminal
report		informs you if the number of lines modified by the last command is greater than the value of the *report* variable
scroll		command mode lines
sections	**sect**	macro names that start sections
shiftwidth	**sw**	for < >, and input ^D
showmatch	**sm**	to) and } as typed
showmode	**smd**	show insert mode in *vi*
slowopen	**slow**	stop updates during insert
term		specifies to **vi** the type of terminal being used (the default is the value of the environmental variable TERM)

window		visual mode lines
wrapmargin	**wm**	automatic line splitting
wrapscan	**ws**	search around end (or beginning) of buffer

Scanning pattern formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any character in *str*
[^*str*]	any character not in *str*
[*x*–*y*]	any character between *x* and *y*
*	any number of preceding characters

AUTHOR

vi and *ex* are based on software developed by the University of California, Berkeley, California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/usr/lib/exstrings	error messages
/usr/lib/exrecover	recover command
/usr/lib/expreserve	preserve command
/usr/lib/terminfo/*	describes capabilities of terminals
$HOME/.exrc	editor startup file
./.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve/*login*	preservation directory (where *login* is the user's login)

NOTES

Several options, although they continue to be supported, have been replaced in the documentation by options that follow the Command Syntax Standard (see **intro**(1)). The **–** option has been replaced by **–s**, a **–r** option that is not followed with an option-argument has been replaced by **–L**, and **+***command* has been replaced by **–c** *command*.

SEE ALSO

crypt(1), ed(1), edit(1), grep(1), sed(1), sort(1), vi(1)
curses(3X), in the *Programmer's Reference Manual.*
term(4), terminfo(4) in the *System Administrator's Reference Manual.*
User's Guide.
"curses/terminfo" chapter of the *Programmer's Guide*.

WARNINGS

The encryption options and commands are provided as a separate package only to source product customers in the United States.

BUGS

The z command prints the number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors do not print a name if the command line –s option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

NULL characters are discarded in input files and cannot appear in resultant files.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

expr arguments

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to the shell must be escaped. Note that **0** is returned to indicate a zero value, rather than the **NULL** string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2s complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within {} symbols.

expr **\|** *expr*
returns the first *expr* if it is neither **NULL** nor **0**, otherwise, returns the second *expr*.

expr **\&** *expr*
returns the first *expr* if neither *expr* is **NULL** or **0**, otherwise, returns **0**.

expr { =, \>, \>=, \<, \<=, != } *expr*
returns the result of an integer comparison if both arguments are integers, otherwise, returns the result of a lexical comparison.

expr { +, – } *expr*
addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
multiplication, division, or remainder of the integer-valued arguments.

expr : *expr*
The matching operator : compares the first argument with the second argument which must be a regular expression. Regular expression syntax is the same as that of *ed*(1), except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (**0** on failure). Alternatively, the **\(...\)** pattern symbols can be used to return a portion of the first argument.

EXAMPLES

1. a='expr $a + 1'

   adds 1 to the shell variable **a**.

2. # 'For $a equal to either "/usr/abc/file" or just "file"'
   expr $a : '.*/\(.*\)' \| $a

   returns the last segment of a pathname (i.e., file). Watch out for / alone as an argument: *expr* will take it as the division operator (see BUGS below).

3. # A better representation of example 2.
   expr //$a : '.*/\(.*\)'

   The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. expr $VAR : '.*'

   returns the number of characters in $VAR.

SEE ALSO

ed(1), sh(1).

DIAGNOSTICS

As a side effect of expression evaluation, *expr* returns the following exit values:

0	if the expression is neither **NULL** nor **0**
1	if the expression *is* **NULL** or **0**
2	for invalid expressions.
syntax error	for operator/operand errors
non-numeric argument	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If $a is an =, the command:

```
expr $a = '='
```

looks like:

```
expr = = =
```

as the arguments are passed to *expr* (and they will all be taken as the operator). The following works:

```
expr X$a = X=
```

NAME

factor – obtain the prime factors of a number

SYNOPSIS

factor [*integer*]

DESCRIPTION

When you use *factor* without an argument, it waits for you to give it an integer. After you give it a positive integer less than or equal to 10^{14}, it factors the integer, prints its prime factors the proper number of times, and then waits for another integer. *factor* exits if it encounters a zero or any non-numeric character.

If you invoke *factor* with an argument, it factors the integer as described above, and then it exits.

The maximum time to factor an integer is proportional to $\sqrt{n}$. *factor* will take this time when n is prime or the square of a prime.

DIAGNOSTICS

factor prints the error message, Ouch, for input out of range or for garbage input.

NAME

fgrep – search a file for a character string

SYNOPSIS

fgrep [*options*] *string* [*file* ...]

DESCRIPTION

fgrep (fast *grep*) searches files for a character string and prints all lines that contain that string. *fgrep* is different from *grep(1)* and *egrep(1)* because it searches for a string, instead of searching for a pattern that matches an expression. It uses a fast and compact algorithm.

The characters **$**, *, **[**, ^, **|**, **(**, **)**, and **** are interpreted literally by *fgrep*, that is, *fgrep* does not recognize full regular expressions as does *egrep*. Since these characters have special meaning to the shell, it is safest to enclose the entire *string* in single quotes **'...'**.

If no files are specified, *fgrep* assumes standard input. Normally, each line found is copied to the standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

–b
Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).

–c
Print only a count of the lines that contain the pattern.

–i
Ignore upper/lowercase distinction during comparisons.

–l
Print the names of files with matching lines once, separated by newlines. It does not repeat the names of files when the pattern is found more than once.

–n
Precede each line by its line number in the file (first line is 1).

–v
Print all lines except those that contain the pattern.

–x
Print only lines matched entirely.

–e *special_string*
Search for a *special string* (*string* begins with a –).

–f *file*
Take the list of *strings* from *file*.

SEE ALSO

ed(1), egrep(1), grep(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep* command, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h**.

NAME

file – determine file type

SYNOPSIS

file [**–c**] [**–f** *ffile*] [**–m** *mfile*] *arg* ...

DESCRIPTION

file performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable **a.out**, *file* will print the version stamp, provided it is greater than 0.

–c
The –c option causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under **–c**.

–f
If the **–f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

–m
The **–m** option instructs *file* to use an alternate magic file.

file uses the file **/etc/magic** to identify files that have some sort of *magic number*, i.e., any file containing a numeric or string constant that indicates its type. Commentary at the beginning of **/etc/magic** explains its format.

FILES

/etc/magic

SEE ALSO

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

NAME

find – find files

SYNOPSIS

find *path-name-list expression*

DESCRIPTION

find recursively descends the directory hierarchy for each pathname in the *path-name-list* (that is, one or more pathnames), seeking files that match a boolean *expression* written in the primaries given below. *find* does *not* follow symbolic links to the resulting file or directory. Instead, it applies the selection criteria to the symbolic link itself. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, –*n* means less than *n* and *n* means exactly *n*. Valid expressions are:

–name *file*
True if *file* matches the current file name. Normal shell argument syntax may be used if escaped (watch out for **[**, **?** and *****).

[–perm] *–onum*
True if file-permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, only the bits that are set in *onum* are compared with the file permission flags, and the expression evaluates true if they match.

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

–links *n*
True if the file has *n* links.

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

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

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

–atime *n*
True if the file has been accessed in *n* days. The access time of directories in *path-name-list* is changed by *find* itself.

–mtime *n*
True if the file has been modified in *n* days.

–ctime *n*
True if the file has been changed in *n* days.

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

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

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

–cpio *device*
Always true; write the current file on *device* in *cpio* (1) format (5120-byte records).

–newer *file*
True if the current file has been modified more recently than the argument *file*.

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

–mount
Always true; restricts the search to the file system containing the directory specified, or if no directory was specified, the current directory.

–local
True if the file physically resides on the local system.

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

The primaries may be combined using the following operators, in order of decreasing precedence:

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

EXAMPLE

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

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

FILES

/etc/passwd, /etc/group

SEE ALSO

chmod(1), cpio(1), sh(1), test(1)
stat(2), umask(2), fs(4) in the *Programmer's Reference Manual.*

BUGS

find / –depth always fails with the message:

find: stat failed: : No such file or directory

NAME

fmt – disk initializer

SYNOPSIS

fmt [*options*] *alias*

DESCRIPTION

fmt(1) checks to see that *alias* is in the **permissions** file and that format permission is given. *fmt* takes the same options as *dinit*(1M). Options are added to those found in the *format_pgm* field of the appropriate **permissions** file entry. The *format_pgm* is then executed with these options on slice 7 of the raw device corresponding to the *slice* entry in the **permissions** file.

If no *alias* is specified, the **floppy** alias is assumed.

FILES

/etc/ddefs	
/etc/dskdefs/*	disk definition files
/etc/dinit	
/etc/filesys	permissions file

SEE ALSO

dinit(1M) in the *System Administrator's Reference Manual.*
filesys(4) in the *Programmer's Reference Manual.*

NAME

fs – construct a file system

SYNOPSIS

fs [*disk* [*blocks*[**:***inodes*]]]

DESCRIPTION

fs(1) builds a file system with a single empty directory on it. The argument *disk* and the **permissions** file are used to determine the device to build a file system on. This device will be the first match of *disk* and the *slice* or *alias* entries in the **permissions** file. If the *disk* argument is not given, then the first *alias* of **floppy** in the **permissions** file will be used. *fs* actually uses the raw version of the listed *slice* (by prepending an **r** to the name).

The size of the file system is the value of *blocks* interpreted as a decimal number. This is the number of *physical* disk blocks the file system occupies. This value may not be larger than the default value specified in the **permissions** file. If the number of blocks is not specified, the default value in the **permissions** file is used. The boot program is left uninitialized. If the optional number of inodes is not given, the default is the number of logical blocks divided by four.

FILES

/etc/mkfs
/etc/filesys permissions file

SEE ALSO

mkfs(1M) in the *System Administrator's Reference Manual.*
dir(4), fs(4), filesys(4) in the *Programmer's Reference Manual.*
System Administrator's Guide.

NAME

get – get a version of an SCCS file

SYNOPSIS

get [**–r***SID*] [**–c***cutoff*] [**–i***list*] [**–x***list*] [**–w***string*] [**–a***seq-no.*] [**–k**] [**–e**] [**–l**[p]] [**–p**] [**–m**] [**–n**] [**–s**] [**–b**] [**–g**] [**–t**] *file* ...

DESCRIPTION

get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with **–**. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of **–** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see **FILES**, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

–r*SID*

The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the **–e** keyletter is also used), as a function of the SID specified.

–c*cutoff*

Cutoff date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values, i.e., **–c7502** is equivalent to **–c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows you to specify a *cutoff* date in the

form: **"–c77/2/2 9:22:25"**. Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(1C) command:

~!get "–c%E% %U%" s.file

–i*list*

A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID – SID

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1.

–x*list*

A *list* of deltas to be excluded in the creation of the generated file. See the **–i** keyletter for the *list* format.

–e

Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The **–e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *get*s for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file (see *admin*(1)). Concurrent use of **get –e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an **–e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the **–k** keyletter in place of the **–e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see *admin*(1)) are enforced when the **–e** keyletter is used.

–b

Used with the **–e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file (see *admin*(1)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)

NOTE: A branch *delta* may always be created from a non-leaf *delta*. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

–k
Suppresses replacement of identification keywords (see **–s**) in the retrieved text by their value. The **–k** keyletter is implied by the **–e** keyletter.

–l[p]
Causes a delta summary to be written into an **l-file**. If **–lp** is used then an **l-file** is not created; the delta summary is written on the standard output instead. See **FILES** for the format of the **l-file**.

–p
Causes the text retrieved from the SCCS file to be written on the standard output. No **g-file** is created. All output that normally goes to the standard output goes to file descriptor 2 instead, unless the **–s** keyletter is used, in which case it disappears.

–s
Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

–m
Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

–n
Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **–m** and **–n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **–m** keyletter generated format.

–g
Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an **l-file**, or to verify the existence of a particular SID.

–t
Used to access the most recently created delta in a given release (e.g., **–r1**), or release and level (e.g., **–r1.2**).

–w *string*
Substitute *string* for all occurrences of %W% when getting the file.

–a*seq-no.*
The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter. If both the **–r** and –a keyletters are specified, only the **–a** keyletter is used. Care should be taken when using the **–a** keyletter with the **–e** keyletter, because the SID of the delta to be created may not be what you expects. The **–r** keyletter can be used with the **–a** and **–e** keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **–e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a newline) before it is processed. If the **–i** keyletter is used included deltas are listed following the notation "Included"; if the **–x** keyletter is used, excluded deltas are listed following the notation "Excluded".

Table 1. Determination of SCCS Identification String

SID* Specified	–b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This is used to force creation of the *first* delta in a *new* release.

\# Successor.

† The **–b** keyletter is effective only if the **b** flag (see *admin*(1)) is present in the file. An entry of – means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

DENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	Value
%M%	Module name: either the value of the **m** flag in the file (see *admin*(1)), or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file (see *admin*(1)).
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this should not have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by *what*(1).
%W%	A shorthand notation for constructing *what*(1) strings for UNIX program files. %W%#=#%Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non-UNIX program files. %A% = %Z%%Y% %M% %I%%Z%

Several auxiliary files may be created by *get*. These files are known generically as the **g-file**, **l-file**, **p-file**, and **z-file**. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The **g-file** is an exception to this scheme: the **g-file** is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The **g-file**, which contains the generated text, is created in the current directory (unless the **–p** keyletter is used). A **g-file** is created in all cases, whether any lines of text were generated by the *get*. It is owned by the real user. If the **–k** keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The **l-file** contains a table showing which deltas were applied in generating the retrieved text. The **l-file** is created in the current directory if the **–l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the **l-file** have the following format:

a. A blank character if the delta was applied; other *.

b. A blank character if the delta was applied or was not applied and ignored; * if the delta was not applied and was not ignored.

c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included.
   "X": Excluded.
   "C": Cut off (by a **–c** keyletter).

d. Blank.

e. SCCS identification (SID).

f. Tab character.

g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.

h. Blank.

i. Login name of person who created *delta*.

The comments and **MR** data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The **p-file** is used to pass information resulting from a *get* with an **–e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an **–e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, (see *admin*(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the **p-file** is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the **–i** keyletter argument if it was present, followed by a blank and the **–x** keyletter argument if it was present, followed by a newline. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The **z-file** serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The **z-file** is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the **p-file** apply for the **z-file**. The **z-file** is created mode 444.

FILES

g-file	Existed before the execution of *delta*; removed after completion of *delta*.
p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

admin(1), delta(1), prs(1), what(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user does not, then only one file may be named when the **–e** keyletter is used.

NAME

getopt – parse command options

SYNOPSIS

set — `getopt** *optstring* **$*`

DESCRIPTION

WARNING: Start using the new command *getopts*(1) in place of *getopt*(1). *getopt*(1) will not be supported in the next major release. For more information, see the **WARNINGS** section.

getopt is used to break up options in command lines for easy parsing by shell procedures and to check for legal options. *optstring* is a string of recognized option letters (see *getopt*(3C)); if a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. The special option — is used to delimit the end of the options. If it is used explicitly, *getopt* will recognize it; otherwise, *getopt* will generate it; in either case, *getopt* will place it at the end of the options. The positional parameters ($1 $2 . . .) of the shell are reset so that each option is preceded by a – and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

EXAMPLE

The following code fragment shows how you can process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an argument:

```
set -- getopt abo: $*
if [ $? != 0 ]
then
        echo $USAGE
        exit 2
fi
for i in $*
do
        case $i in
        -a | -b)    FLAG=$i; shift;;
        -o)         OARG=$2; shift 2;;
        --)         shift; break;;
        esac
done
```

This code will accept any of the following as equivalent:

```
cmd -aoarg file file
cmd -a -o arg file file
cmd -oarg -a file file
cmd -a -oarg -- file file
```

SEE ALSO

getopts(1), sh(1)
getopt(3C) in the *Programmer's Reference Manual.*

DIAGNOSTICS

getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

WARNINGS

getopt (1) does not support the part of Rule 8 of the command syntax standard (see *intro*(1)) that permits groups of option-arguments following an option to be separated by white space and quoted. For example,

```
cmd -a -b -o "xxx z yy" file
```

is not handled correctly). To correct this deficiency, use the new command *getopts* (1) in place of *getopt* (1).

getopt (1) will not be supported in the next major release. For this release a conversion tool has been provided, *getoptcvt*. For more information about *getopts* and *getoptcvt*, see the *getopts* (1) manual page.

If an option that takes an option-argument is followed by a value that is the same as one of the options listed in *optstring* (referring to the earlier **EXAMPLE** section, but using the following command line:

cmd -o -a file

getopt will always treat **–a** as an option-argument to **–o**; it will never recognize **–a** as an option. For this case, the **for** loop in the example will shift past the *file* argument.

NAME

getopts, getoptcvt – parse command options

SYNOPSIS

getopts *optstring name* [*arg ...*]

/usr/lib/getoptcvt [**–b**] *file*

DESCRIPTION

getopts is used by shell procedures to parse positional parameters and to check for legal options. It supports all applicable rules of the command syntax standard (see Rules 3-10, *intro*(1)). It should be used in place of the *getopt*(1) command (see **WARNING**).

optstring must contain the option letters the command using *getopts* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

Each time it is invoked, *getopts* will place 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 or a shell procedure is invoked, OPTIND is initialized to **1**.

When an option requires an option-argument, *getopts* places it in the shell variable OPTARG .

If an illegal option is encountered, **?** will be placed in *name*.

When the end of options is encountered, *getopts* exits with a non-zero exit status. The special option "—" may be used to delimit the end of the options.

By default, *getopts* parses the positional parameters. If extra arguments (*arg ...*) are given on the *getopts* command line, *getopts* will parse them instead.

/usr/lib/getoptcvt reads the shell script in *file*, converts it to use *getopts*(1) instead of *getopt*(1), and writes the results on the standard output.

–b

the results of running **/usr/lib/getoptcvt** will be portable to earlier releases of the UNIX system. **/usr/lib/getoptcvt** modifies the shell script in *file* so that when the resulting shell script is executed, it determines at run time whether to invoke *getopts*(1) or *getopt*(1).

So all new commands will adhere to the command syntax standard described in *intro*(1), they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command (see **WARNINGS**).

EXAMPLE

The following fragment of a shell program shows how you can process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an option-argument:

```
while getopts abo: c
do
        case $c in
        a | b)      FLAG=$c;;
        o)          OARG=$OPTARG;;
        \?)         echo $USAGE
                    exit 2;;
        esac
done
shift   expr $OPTIND - 1
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o "xxx z yy" file
cmd -a -b -o "xxx z yy" -- file
cmd -ab -o xxx,z,yy file
cmd -ab -o "xxx z yy" file
cmd -o xxx,z,yy -b -a file
```

SEE ALSO

intro(1), sh(1)
getopts(3C) in the *Programmer's Reference Manual*.

WARNING

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the **EXAMPLE** section, **a** and **b** are options, and the option **o** requires an option-argument:

cmd –aboxxx file (Rule 5 violation: options with option-arguments must not be grouped with other options)
cmd –ab –oxxx file (Rule 6 violation: there must be white space after an option that takes an option-argument)

Changing the value of the shell variable OPTIND or parsing different sets of arguments may lead to unexpected results.

DIAGNOSTICS

getopts prints an error message on the standard error when it encounters an option letter not included in *optstring*.

NAME

glossary – definitions of terms and symbols

SYNOPSIS

[**help**] **glossary** [*term*]

DESCRIPTION

The Help Facility command *glossary* provides definitions of common technical terms and symbols.

Without an argument, *glossary* displays a menu screen listing the terms and symbols that are currently included in *glossary*. A user may choose one of the terms or may exit to the shell by typing q (for "quit"). When a term is selected, its definition is retrieved and displayed. By selecting the appropriate menu choice, the list of terms and symbols can be redisplayed.

A term's definition may also be requested directly from shell level (as shown above), causing a definition to be retrieved and the list of terms and symbols not to be displayed. Some of the symbols must be escaped if requested at shell level in order for the facility to understand the symbol. The following is a table that lists the symbols and their escape sequence.

Symbol	Escape Sequence
""	\"\"
''	\'\'
[]	\\[\\]
‘’	\'\'
#	\#
&	\&
*	*
\	\\\\
|	\\|

From any screen in the Help Facility, a user may execute a command via the shell (*sh*(1)) by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable SCROLL must be set to **no** and exported so it will become part of your environment. This is done by adding the

following line to your **.profile** file (see *profile*(4)): "export SCROLL ; SCROLL=no". If you later decide that scrolling is desired, SCROLL must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

SEE ALSO

help(1), helpadm(1M), locate(1), sh(1), starter(1), usage(1)
term(5) in the *Programmer's Reference Manual*.

WARNINGS

If the shell variable TERM (see *sh*(1)) is not set in the user's **.profile** file, then TERM will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

greek – select terminal filter

SYNOPSIS

greek [**–T***terminal*]

DESCRIPTION

greek is a filter that reinterprets the extended character set, as well as the reverse and half-line motions, of a 128-character Teletype Model 37 terminal for certain other terminals. Special characters are simulated by overstriking, if necessary and possible. If the argument is omitted, *greek* attempts to use the environment variable $TERM (see *environ*(5)). Currently, the following *terminal*s are recognized:

300	DASI 300.
300-12	DASI 300 in 12-pitch.
300s	DASI 300s.
300s-12	DASI 300s in 12-pitch.
450	DASI 450.
450-12	DASI 450 in 12-pitch.
1620	Diablo 1620 (alias DASI 450).
1620-12	Diablo 1620 (alias DASI 450) in 12-pitch.
2621	Hewlett-Packard 2621, 2640, and 2645.
2640	Hewlett-Packard 2621, 2640, and 2645.
2645	Hewlett-Packard 2621, 2640, and 2645.
4014	Tektronix 4014.
hp	Hewlett-Packard 2621, 2640, and 2645.
tek	Tektronix 4014.

FILES

/usr/bin/300
/usr/bin/300s
/usr/bin/4014
/usr/bin/450
/usr/bin/hp

SEE ALSO

300(1), 4014(1), 450(1), hp(1), tplot(1G).
eqn(1), mm(1), nroff(1) in the *DOCUMENTER'S WORKBENCH Software Release 2.0 Technical Discussion and Reference Manual.*
environ(5), greek(5), term(5) in the *Programmer's Reference Manual.*

NAME

grep – search a file for a pattern

SYNOPSIS

grep [*options*] *limited regular expression* [*file ...*]

DESCRIPTION

grep searches files for a pattern and prints all lines that contain that pattern. *grep* uses limited regular expressions (expressions that have string values that use a subset of the possible alphanumeric and special characters) like those used with *ed*(1) to match the patterns. It uses a compact non-deterministic algorithm.

Be careful using the characters **$**, *, [, ^, |, (,), and \ in the *limited regular expression* because they are also meaningful to the shell. It is safest to enclose the entire *limited regular expression* in single quotes **'...'**.

If no files are specified, *grep* assumes standard input. Normally, each line found is copied to standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

–b
Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).

–c
Print only a count of the lines that contain the pattern.

–i
Ignore upper/lowercase distinction during comparisons.

–l
Print the names of files with matching lines once, separated by newlines. Does not repeat the names of files when the pattern is found more than once.

–n
Precede each line by its line number in the file (first line is 1).

–s
Suppress error messages about nonexistent or unreadable files

–v
Print all lines except those that contain the pattern.

SEE ALSO

ed(1), egrep(1), fgrep(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h**.

If there is a line with embedded **NULL**s, *grep* will only match up to the first **NULL**; if it matches, it will print the entire line.

NAME

help – Help Facility

SYNOPSIS

help
[**help**] **starter**
[**help**] **usage** [**–d**] [**–e**] [**–o**] [*command_name*]
[**help**] **locate** [*keyword1* [*keyword2*] ...]
[**help**] **glossary** [*term*]
help *arg* ...

DESCRIPTION

The Help Facility provides on-line assistance for users who desire general information or specific assistance with the SCCS commands.

Without arguments, *help* prints a menu of available on-line assistance commands with a short description of their functions. The commands and their descriptions are:

Command	Description
starter	information about the system for the beginning user
locate	locate commands using function-related keywords
usage	command usage information
glossary	definitions of technical terms

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

With arguments, *help* directly invokes the named online assistance command, bypassing the initial *help* menu. The commands *starter*, *locate*, *usage*, and *glossary*, optionally preceded by the word *help*, may also be specified at shell level. When executing *glossary* from shell level some of the symbols listed in the glossary must be escaped (preceded by one or more backslashes, "\ to be understood by the Help Facility. For a list of symbols and how many backslashes to use for each, refer to the *glossary*(1) manual page.

From any screen in the Help Facility, a user may execute a command via the shell (*sh*(1)) by typing a **!** and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable SCROLL must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your **.profile** file (see *profile*(4)): "export SCROLL ; SCROLL=no". If you later decide that scrolling is desired, SCROLL must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

The Help Facility can be tailored to a customer's needs by use of the *helpadm*(1M) command.

If the first argument to *help* is different from *starter*, *usage*, *locate*, or *glossary*, *help* assumes information is being requested about the SCCS Facility. The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type1
: Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., ge3 for message 3 from the *get* command).

type2
: Does not contain numerics (as a command, e.g., *get*).

type3
: Is all numeric (e.g., 212).

SEE ALSO

glossary(1), helpadm(1M), locate(1), sh(1), starter(1), usage(1)
admin(1), cdc(1), comb(1), delta(1), get(1), prs(1), rmdel(1), sact(1), sccsdiff(1), unget(1), val(1), vc(1), what(1), profile(4), sccsfile(4), term(5) in the *Programmer's Reference Manual*.

WARNINGS

If the shell variable TERM (see *sh*(1)) is not set in the user's environment (see *environment*(5)) file, TERM defaults to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

hp – handle special functions of Hewlett-Packard terminals

SYNOPSIS

hp [–e] [–m]

DESCRIPTION

hp supports special functions of the Hewlett-Packard 2640 series of terminals, with the primary purpose of producing accurate representations of most *nroff* output. In the following discussion it should be noted that, unless your system contains the DOCUMENTER'S WORKBENCH Software, references to certain commands (e.g., ***nroff, neqn, equ,*** etc.) will not work. A typical usage is in conjunction with DOCUMENTER'S WORKBENCH Software:

nroff –h files ... | hp

Regardless of the hardware options on your terminal, *hp* tries to do sensible things with underlining and reverse linefeeds. If the terminal has the "display enhancements" feature, subscripts and superscripts can be indicated in distinct ways. If it has the "mathematical-symbol" feature, Greek and other special characters can be displayed.

The flags are:

–e

It is assumed that your terminal has the "display enhancements" feature, and so maximal use is made of the added display modes. Overstruck characters are presented in the Underline mode. Superscripts are shown in Half-bright mode, and subscripts in Half-bright, Underlined mode. If this flag is omitted, *hp* assumes that your terminal lacks the "display enhancements" feature. In this case, all overstruck characters, subscripts, and superscripts are displayed in Inverse Video mode, i.e., dark-on-light, rather than the usual light-on-dark.

–m

Requests minimization of output by removal of newlines. Any contiguous sequence of 3 or more newlines is converted into a sequence of only 2 newlines; i.e., any number of successive blank lines produces only a single blank output line. This allows you to retain more actual text on the screen.

With regard to Greek and other special characters, *hp* provides the same set as does *300*(1), except that "not" is approximated by a right arrow, and only the top half of the integral sign is shown.

DIAGNOSTICS

"line too long" if the representation of a line exceeds 1,024 characters.
The exit codes are **0** for normal termination, **2** for all errors.

SEE ALSO

300(1), greek(1).

BUGS

An "overstriking sequence" is defined as a printing character followed by a backspace followed by another printing character. In such sequences, if either printing character is an underscore, the other printing character is shown underlined or in Inverse Video; otherwise, only the first printing character is shown (again, underlined or in Inverse Video). Nothing special is done if a **BACKSPACE** is adjacent to an ASCII control character. Sequences of control characters (e.g., reverse linefeeds, **BACKSPACES**) can make text "disappear"; in particular, tables generated by *tbl*(1) that contain vertical lines will often be missing the lines of text that contain the "foot" of a vertical line, unless the input to *hp* is piped through *col*(1).

Although some terminals do provide numerical superscript characters, no attempt is made to display them.

NAME

hpio – Hewlett-Packard 2645A terminal tape file archiver

SYNOPSIS

hpio –o[rc] *file ...*

hpio –i[rta] [**–n** *count*]

DESCRIPTION

hpio is designed to take advantage of the tape drives on Hewlett-Packard 2645A terminals. Up to 255 SYSTEM V/88 files can be archived onto a tape cartridge for offline storage or for transfer to another SYSTEM V/88 system. The actual number of files depends on the sizes of the files. One file of about 115,000 bytes will almost fill a tape cartridge. Almost 300 1-byte files will fit on a tape, but the terminal will not be able to retrieve files after the first 255. This manual page is not intended to be a guide for using tapes on Hewlett-Packard 2645A terminals, but tries to give enough information to be able to create and read tape archives and to position a tape for access to a desired file in an archive.

hpio **–o** (copy out) copies the specified *file*(s), together with pathname and status information to a tape drive on your terminal (which is assumed to be positioned at the beginning of a tape or immediately after a tape mark). The left tape drive is used by default. Each *file* is written to a separate tape file and terminated with a tape mark. When *hpio* finishes, the tape is positioned following the last tape mark written.

hpio **–i** (copy in) extracts a file(s) from a tape drive (which is assumed to be positioned at the beginning of a file that was previously written by a *hpio* **–o**). The default action extracts the next file from the left tape drive.

hpio always leaves the tape positioned after the last file read from or written to the tape. Tapes should always be rewound before the terminal is turned off. To rewind a tape depress the green function button, then function key 5, and then select the appropriate tape drive by depressing either function key 5 for the left tape drive or function key 6 for the right. If several files have been archived onto a tape, the tape may be positioned at the beginning of a specific file by depressing the green function button, then function key 8, followed by typing the desired file number (1–255) with no RETURN, and finally function key 5 for the left tape or function key 6 for the right. The desired file number may also be specified by a signed number relative to the current file number.

The meanings of the available options are:

r
Use the right tape drive.

c
Include a checksum at the end of each *file*. The checksum is always checked by *hpio* **–i** for each file written with this option by *hpio* **–o**.

n *count*
The number of input files to be extracted is set to *count*. If this option is not given, *count* defaults to 1. An arbitrarily large *count* may be specified to extract all files from the tape. *hpio* will stop at the end of data mark on the tape.

t
Print a table of contents only. No files are created. Printed information gives the file size in bytes, the file name, the file access modes, and whether or not a checksum is included for the file.

a
Ask before creating a file. *hpio*–i normally prints the file size and name, creates and reads in the file, and prints a status message when the file has been read in. If a checksum is included with the file, it reports whether the checksum matched its computed value. With this option, the file size and name are printed followed by a **?**. Any response beginning with **y** or **Y** will cause the file to be copied in as above. Any other response will cause the file to be skipped.

FILES

/dev/tty?? to block messages while accessing a tape

SEE ALSO

cu(1C).

DIAGNOSTICS

BREAK
An interrupt signal terminated processing.

Can't create '*file*'.
File system access permissions did not allow *file* to be created.

Can't get tty options on stdout.
hpio was unable to get the input-output control settings associated with the terminal.

Can't open '*file*'.
file could not be accessed to copy it to tape.

End of Tape.
No tape record was available when a read from a tape was requested. An end of data mark is the usual reason for this, but it may also occur if the wrong tape drive is being accessed and no tape is present.

'*file*' not a regular file.
file is a directory or other special file. Only regular files will be copied to tape.

Readcnt = *rc*, termcnt = *tc*.
hpio expected to read *rc* bytes from the next block on the tape, but the block contained *tc* bytes. This is caused by having the tape improperly positioned or by a tape block being mangled by interference from other terminal I/O.

Skip to next file failed.
An attempt to skip over a tape mark failed.

Tape mark write failed.
An attempt to write a tape mark at the end of a file failed.

Write failed.
A tape write failed. This is most frequently caused by specifying the wrong tape drive, running off the end of the tape, or trying to write on a tape that is write protected.

WARNINGS

Tape I/O operations may copy bad data if any other I/O involving the terminal occurs. Do not attempt any type ahead while *hpio* is running. *hpio* turns off write permissions for other users while it is running, but processes started asynchronously from your terminal can still interfere. The most common indication of this problem, while a tape is being written, is the appearance of characters on the display screen that should have been copied to tape.

The keyboard, including the terminal **BREAK** key, is locked during tape write operations; the **BREAK** key is only functional between writes.

hpio must have complete control of the attributes of the terminal to communicate with the tape drives. Interaction with commands such as *cu*(1C) may interfere and prevent successful operation.

BUGS

Some binary files contain sequences that will confuse the terminal.

An *hpio* **–i** that encounters the end of data mark on the tape (e.g., scanning the entire tape with *hpio –itn 300*), leaves the tape positioned *after* the end of data mark. If a subsequent *hpio* **–o** is done at this point, the data

will not be retrievable. The tape must be repositioned manually using the terminal FIND FILE –1 operation (depress the green function button, function key 8, and then function key 5 for the left tape or function key 6 for the right tape) before the *hpio* **–o** is started.

If an interrupt is received by *hpio* while a tape is being written, the terminal may be left with the keyboard locked. If this happens, the terminal's **RESET TERMINAL** key will unlock the keyboard.

NAME

hyphen – find hyphenated words

SYNOPSIS

hyphen [*files*]

DESCRIPTION

hyphen finds all the hyphenated words ending lines in *files* and prints them on the standard output. If no arguments are given, the standard input is used; thus, *hyphen* may be used as a filter.

EXAMPLES

The following allows the proofreading of *nroff*s hyphenation in *textfile*:

nroff -mm textfile | hyphen

(Note that *nroff* is not a supported product on SYSTEM V/88, and is used here only for an example.)

BUGS

hyphen cannot cope with hyphenated *italic* (i.e., underlined) words; it will often miss them completely, or mangle them.

hyphen occasionally gets confused, but with no ill effects other than spurious extra output.

NAME

ipcrm – remove a message queue, semaphore set or shared memory id

SYNOPSIS

ipcrm [*options*]

DESCRIPTION

ipcrm will remove one or more specified messages, semaphore or shared memory identifiers. The identifiers are specified by the following *options*:

–q *msqid*
removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.

–m *shmid*
removes the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

–s *semid*
removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.

–Q *msgkey*
removes the message queue identifier, created with key *msgkey*, from the system and destroys the message queue and data structure associated with it.

–M *shmkey*
removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

–S *semkey*
removes the semaphore identifier, created with key *semkey*, from the system and destroys the set of semaphores and data structure associated with it.

The details of the removes are described in *msgctl*(2), *shmctl*(2), and *semctl*(2). The identifiers and keys may be found by using *ipcs*(1).

SEE ALSO

ipcs(1).
msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2) in the *Programmer's Reference Manual*.

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

ipcs [*options*]

DESCRIPTION

ipcs prints certain information about active inter-process communication facilities. Without *options*, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following *options*:

–q
Print information about active message queues.

–m
Print information about active shared memory segments.

–s
Print information about active semaphores.

If any of the options **–q**, **–m**, or **–s** are specified, information about only those indicated will be printed. If none of these three are specified, information about all three will be printed subject to these options:

–b
Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.) See below for meaning of columns in a listing.

–c
Print creator's login name and group name. See below.

–o
Print information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments.)

–p
Print process number information. (Process ID of last process to send a message and process ID of last process to receive a message on message queues and process ID of creating process and process ID of last process to attach or detach on shared memory segments.) See below.

–t
Print time information. (Time of the last control operation that changed the access permissions for all facilities. Time of last *msgsnd* and last *msgrcv* on message queues, last *shmat* and last *shmdt* on shared memory, last *semop*(2) on semaphores.) See below.

–a
Use all print *options*. (This is a shorthand notation for **–b**, **–c**, **–o**, **–p**, and **–t**.)

–C *corefile*
Use the file *corefile* in place of **/dev/kmem**.

–N *namelist*
Use the file *namelist* in place of **/unix**.

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. Note that these *options* only determine what information is provided for each facility; they do *not* determine which facilities will be listed.

T	(all)	Type of the facility: **q** message queue; **m** shared memory segment; **s** semaphore.
ID	(all)	The identifier for the facility entry.
KEY	(all)	The key used as an argument to *msgget*, *semget*, or *shmget* to create the facility entry. (NOTE: The key of a shared memory segment is changed to **IPC_PRIVATE** when the segment has been removed until all processes attached to the segment detach it.)

MODE (all) The facility access modes and flags: The mode consists of 11 characters that are interpreted as:
The first two characters are:

- **R** if a process is waiting on a *msgrcv*;
- **S** if a process is waiting on a *msgsnd*;
- **D** if the associated shared memory segment has been removed. It will disappear when the last process attached to the segment detaches it;
- **C** if the associated shared memory segment is to be cleared when the first attach is executed;
- **–** if the corresponding special flag is not set.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. 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 indicated as:

- **r** if read permission is granted;
- **w** if write permission is granted;
- **a** if alter permission is granted;
- **–** if the indicated permission is *not* granted.

OWNER (all) The login name of the owner of the facility entry.

GROUP (all) The group name of the group of the owner of the facility entry.

CREATOR (a,c) The login name of the creator of the facility entry.

CGROUP (a,c) The group name of the group of the creator of the facility entry.

CBYTES (a,o) The number of bytes in messages currently outstanding on the associated message queue.

QNUM (a,o) The number of messages currently outstanding on the associated message queue.

QBYTES (a,b) The maximum number of bytes allowed in messages outstanding on the associated message queue.

LSPID (a,p) The process ID of the last process to send a message to the associated queue.

LRPID	(a,p)	The process ID of the last process to receive a message from the associated queue.
STIME	(a,t)	The time the last message was sent to the associated queue.
RTIME	(a,t)	The time the last message was received from the associated queue.
CTIME	(a,t)	The time when the associated entry was created or changed.
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 entry.
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.
NSEMS	(a,b)	The number of semaphores in the set associated with the semaphore entry.
OTIME	(a,t)	The time the last semaphore operation was completed on the set associated with the semaphore entry.

FILES

/unix	system namelist
/dev/kmem	memory
/etc/passwd	user names
/etc/group	group names

WARNING

If the user specifies either the –C or –N flag, the real and effective UID/GID will be set to the real UID/GID of the user invoking *ipcs*.

SEE ALSO

msgop(2), semop(2), shmop(2) in the *Programmer's Reference Manual*.

BUGS

Things can change while *ipcs* is running; the picture it gives is only a close approximation to reality.

A restricted shell can be invoked in one of the following ways: (1) *rsh* is the file name part of the last entry in the **/etc/passwd** file (see *passwd*(4)); (2) the environment variable SHELL exists and *rsh* is the file name part of its value; (3) the shell is invoked and *rsh* is the file name part of argument 0; (4) the shell is invoke with the **–r** option.

When a command to be executed is found to be a shell procedure, *rsh* invokes *sh* to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the **.profile** (see *profile*(4)) has complete control over user actions by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (i.e., **/usr/rbin**) that can be safely invoked by a restricted shell. Some systems also provide a restricted editor, *red*.

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the *exit* command above).

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

cd(1), echo(1), env(1), getopts(1), intro(1), login(1), newgrp(1), pwd(1), test(1), umask(1), wait(1)
dup(2), exec(2), fork(2), pipe(2), profile(4), signal(2), ulimit(2) in the *Programmer's Reference Manual.*

CAVEATS

Words used for file names in input/output redirection are not interpreted for file name generation (see **File Name Generation**). For example, **cat file1 >a*** will create a file named **a***.

Because commands in pipelines are run as separate processes, variables set in a pipeline have no effect on the parent shell.

If you get the error message *cannot fork, too many processes*, try using the *wait* (1) command to clean up your background processes. If this does not help, the system process table is probably full or you have too many active foreground processes. (There is a limit to the number of process ids associated with your login, and to the number the system can keep track of.)

BUGS

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the *hash* command to correct this situation.

If you move the current directory or one above it, *pwd* may not give the correct response. Use the *cd* command with a full pathname to correct this situation.

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus cannot be waited for.

For *wait n*, if *n* is not an active process id, all your shell's currently active background processes are waited for and the return code will be zero.

NAME

shl – shell layer manager

SYNOPSIS

shl

DESCRIPTION

shl allows a user to interact with more than one shell from a single terminal. The user controls these shells, known as *layers*, using the commands described below.

The *current layer* is the layer which can receive input from the keyboard. Other layers attempting to read from the keyboard are blocked. Output from multiple layers is multiplexed onto the terminal. To have the output of a layer blocked when it is not current, the *stty* option **loblk** may be set within the layer.

The *stty* character **swtch** (set to ^Z if NUL) is used to switch control to *shl* from a layer. *shl* has its own prompt, **>>>**, to help distinguish it from a layer.

A *layer* is a shell which has been bound to a virtual tty device (**/dev/sxt???**). The virtual device can be manipulated like a real tty device using *stty*(1) and *ioctl*(2). Each layer has its own process group id.

Definitions

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. They are used by *shl* when no name is supplied. They may be abbreviated to just the digit.

Commands

The following commands may be issued from the *shl* prompt level. Any unique prefix is accepted.

create [*name*]
Create a layer called *name* and make it the current layer. If no argument is given, a layer will be created with a name of the form *(#)* where # is the last digit of the virtual device bound to the layer. 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* ...]
For each *name*, block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option **–loblk** within the layer.

delete *name* [*name* ...]
For each *name*, delete the corresponding layer. All processes in the process group of the layer are sent the SIGHUP signal (see *signal*(2)).

help (or **?**)
Print the syntax of the *shl* commands.

layers [**–l**] [*name* ...]
For each *name*, list the layer name and its process group. The **–l** option produces a *ps*(1)-like listing. If no arguments are given, information is presented for all existing layers.

resume [*name*]
Make the layer referenced by *name* the current layer. If no argument is given, the last existing current layer will be resumed.

toggle
Resume the layer that was current before the last current layer.

unblock *name* [*name* ...]
For each *name*, do not block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the *stty* option **–loblk** within the layer.

quit
Exit *shl*. All layers are sent the SIGHUP signal.

name
Make the layer referenced by *name* the current layer.

FILES

/dev/sxt???	Virtual tty devices
$SHELL	Variable containing pathname of the shell to use (default is /bin/sh).

SEE ALSO

sh(1), stty(1)
ioctl(2), signal(2) in the *Programmer's Reference Manual.*
sxt(7) in the *System Administrator's Reference Manual.*

NAME

sifilter - preprocess MC88100 assembly language

SYNOPSIS

sifilter [*-switches*] [*input*] [*output*]

DESCRIPTION

sifilter manipulates MC88100 assembly language source code from *input* to work around known problems in early silicon. *sifilter* is normally invoked transparently by the shell script **/bin/as,** but can be used interactively for testing purposes. The program will no longer be needed when the final revision of the silicon is available.

input and *output* are normally omitted, defaulting to standard input and output paths. File names may be specified for either path, and a dash (-), denoting standard input, may be used as a place holder for *input*.

The translations performed by *sifilter* are controlled by the *switches* listed below. The shell script **/bin/cc** sets the "standard" option switches. In the released software all silicon filter options are disabled. If the user is generating object for old silicon, then sifilter switches can be enabled either on the command line, by using the **-F** option to the assembler, or by editing **/bin/cc**.

Switches

a
Insert a *trap-not-taken* (tb1 0,r0,511) after each **ld** or **ld.d.**

b
Split each **st.d** into an equivalent sequence of two **st** instructions.

c
Do not pass comment lines through to the output.

d
Issue each **st** or **st.d** twice.

e
Enable literal synthesis. See **Literal Synthesis.** (**NOTE**: This option is not compatible with some of the other options available, its use is no longer supported.)

f Signals that input comes from the Absoft FORTRAN compiler.

g
Signals that input comes from the Greenhills C compiler.

See the **Scratch Registers** section for an explanation of the need for the **f** and **g** switches.

j Fix a problem with certain instances of the immediate forms of *div* and *divu*.

k
Insert a *no-op* (or r0,r0,r0) after each *trap-not-taken* inserted by the **a** option. If the **a** option has not been specified, this option has no effect.

l Split each **ld.d** into an equivalent sequence of two **ld** instructions.

n
Insert a *no-op* after each *trap-not-taken* inserted by the **p** option, but only if the store instruction involved uses scaling. Specifying the **n** option automatically forces the **p** option.

p
Insert a *trap-not-taken* before each **st** or **st.d.**

q
Insert a dummy **ld** before each **ld.** A dummy load is a load in which the destination register is r0. The source operands in a dummy load are the same as those in the actual load which follows.

r Fix various problems with the *def* and *sdef* directives produced by the Greenhills C compiler. This option has no effect with the Motorola C or Absoft FORTRAN compilers.

s
Produce a statistics dump on the standard error path on termination.

t Compensate for bugs in floating-point operations which have a double-precision destination. See **WARNINGS.**

u
Adds a *no-op* after an **lda.d** if it is immediately followed by a **ld.bu.**

V
Displays a version identification message and exits immediately.

v
A single **v** enables "verbose" mode, in which various messages detailing actions taken by *sifilter* are output as comment lines. Two or more instances of **v** in the option string generates a comment line containing the current location counter value before each source line.

w
Compensate for a bug in the *fmul* instruction in rev D.5 MC88100 chips.

x
Compensate for a bug in the *fmul* instruction by breaking certain immediate integer divides and multiplies into register-to-register forms. (This fix only works in conjunction with a kernel exception code fix.)

y
Insert a *no-op* after each *trap-not-taken* generated by the **z** option. If **z** has not been specified, this option has no effect.

z
Insert a trap-not-taken after each **st** or **st.d.**

E
Insert a **tcnd eq0** before all divides. This compensates for a bug in all mask revisions up to and including E.2, where divide by zero does not always trigger a trap.

Defaults

All switches default to "off". If neither of the **g** or **f** switches are specified, the input is assumed to have been generated by the Motorola C compiler.

sifilter performs the following transformations regardless of the option switches specified.

addu and **subu** instructions with operands r31,r31,lit32 where "lit32" is a constant whose value is greater than 64K are replaced with an equivalent sequence.

Floating point instructions involving double operands may be moved if they would otherwise fall at the end of a cache line.

Literal Synthesis

Since *sifilter* must maintain an accurate location counter, it must perform the same fixups for "lit16" operands that would normally be done by *ld*.

Instructions with lit16 operands whose value cannot be determined by *sifilter* (for example, a label), or whose value would require more than 16 bits, are replaced with an equivalent sequence. This is called "literal synthesis", since a 32-bit value is "synthesized" in a register from the literal.

There are two forms of literal synthesis. The short form:

```
or.u  r29,r0,hi16(lit16)
op    rd,r29,lo16(lit16)
```

is used for the **add, addu, ld, lda, or, st, xmem,** and **xor** instructions (in all their variations) when the source register is r0. When the source register is other than r0, these instructions are expanded into the long form:

```
or.u  r29,r0,hi16(lit16)
or    r29,r29,lo16(lit16)
op    rd,rs,r29
```

Instructions which always are expanded with the long form are all the variations on **and, cmp, div, divu, mask, mul, sub, subu,** and **tbnd.**

No literal synthesis is done unless the **e** option has been specified.

Scratch Registers

Some of the fixups performed by *sifilter* require one or two scratch registers (split **ld.d** or **st.d, addu, subu,** and floats).

Fixups which require only one scratch register will always succeed, since all of the supported compilers have at least one available. Fixups requiring two scratch registers will succeed with the Motorola or Greenhills C compilers, but will fail with the Absoft FORTRAN compiler, which has only one available register.

SEE ALSO

/bin/as, /bin/cc

WARNINGS

Code generated when the **-t** switch is in effect requires special kernel support. DO NOT USE the **-t** option unless your operating system kernel is dated 880815 or later. (To examine the kernel date, enter the command **uname -a.** The date will be the fourth field in the output.)

The sifilter cannot deal with labels in branch delay slots if the branch delay slot instruction requires fixup. In these cases the message:

```
FATAL: delay branch instruction has label
```

is written to **stderr** and *sifilter* will exit. This is usually only a problem with hand-optimized assembly source files, which should not require filtering in any case.

BUGS

Given the large number of option switches, and the almost infinite number of instruction combinations, bugs would not be surprising.

NAME

sink – canonical "server" process for testing network.

SYNOPSIS

sink

DESCRIPTION

sink spawns a daemon which listens on the port used by the TCP sink service (see *services*(4)). As connections are made to that port, *sink* reads and discards any incoming data for the duration of the connection. Multiple and simultaneous connections are supported.

FILES

/etc/services determines port number

NAME

size – print section sizes in bytes of common object files

SYNOPSIS

size [**–n**] [**–f**] [**–o**] [**–x**] [**–V**] *files*

DESCRIPTION

The *size* command produces section size information in bytes for each loaded section in the common object files. The size of the text, data, and bss (uninitialized data) sections is printed, as well as the sum of the sizes of these sections. If an archive file is input to the *size* command the information for all archive members is displayed.

The **-n** option includes NOLOAD sections in the size.

The **-f** option produces full output, that is, it prints the size of every loaded section, followed by the section name in parentheses.

Numbers will be printed in decimal unless either the **–o** or the –x option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **–V** flag will supply the version information on the *size* command.

SEE ALSO

as(1), cc(1), ld(1), a.out(4), ar(4)

CAVEAT

Since the size of bss sections is not known until link-edit time, the *size* command will not give the true total size of pre-linked objects.

DIAGNOSTICS

size: name: cannot open
if *name* cannot be read.

size: name: bad magic
if *name* is not an appropriate common object file.

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep *time*

DESCRIPTION

sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time, as in:

(sleep 105; *command*)&

or to execute a command every so often, as in:

```
while true
do
        command
        sleep 37
done
```

SEE ALSO

alarm(2), sleep(3C) in the *Programmer's Reference Manual.*

NAME

sort – sort and/or merge files

SYNOPSIS

sort [**–cmu**] [**–o***output*] [**–y***kmem*] [**–z***recsz*] [**–dfiMnr**] [**–bt***x*]
[**+***pos1* [**–***pos2*]] [*files*]

DESCRIPTION

sort sorts lines of all the named files together and writes the result on the standard output. The standard input is read if **–** is used as a file name or no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

The following options alter the default behavior:

–c
Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

–m
Merge only, the input files are already sorted.

–u
Unique: suppress all but one in each set of lines having equal keys.

–o*output*
The argument given is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. There may be optional blanks between **–o** and *output*.

–y*kmem*
The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is a waste. If this option is omitted, *sort* begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, *kmem*, *sort* will start using that number of kilobytes of memory, unless the administrative minimum or maximum is violated, in which case the corresponding extremum will be used. Thus, **–y0** is guaranteed to start with minimum memory. By convention, **–y** (with no argument) starts with maximum memory.

–z*recsz*
The size of the longest line read is recorded in the sort phase so buffers can be allocated during the merge phase. If the sort phase is omitted via the **–c** or **–m** options, a popular system default size will be used. Lines longer than the buffer size 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.

The following options override the default ordering rules.

–d
"Dictionary" order: only letters, digits, and blanks (spaces and tabs) are significant in comparisons.

–f
Fold lowercase letters into uppercase.

–i
Ignore non-printable characters.

–M
Compare as months. The first three non-blank characters of the field are folded to uppercase and compared. For example, in English the sorting order is "JAN" < "FEB" < . . . < "DEC". Invalid fields compare low to "JAN". The **–M** option implies the **–b** option (see below).

–n
An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The **–n** option implies the **–b** option (see below). Note that the **–b** option is only effective when restricted sort key specifications are in effect.

–r
Reverse the sense of comparisons.

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

The notation *+pos1 –pos2* restricts a sort key to one beginning at *pos1* and ending just before *pos2*. The characters at position *pos1* and just before *pos2* are included in the sort key (provided that *pos2* does not precede *pos1*). A missing *–pos2* means the end of the line.

Specifying *pos1* and *pos2* involves the notion of a field, a minimal sequence of characters followed by a field separator or a newline. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field. The treatment of field separators can be altered using the options:

–b
Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the **–b** option is specified before the first *+pos1* argument, it will be applied to all *+pos1* arguments. Otherwise, the **b** flag may be attached independently to each *+pos1* or *–pos2* argument (see below).

–t*x*
Use *x* as the field separator character; *x* is not considered to be part of a field (although it may be included in a sort key). Each occurrence of *x* is significant (for example, *xx* delimits an empty field).

pos1 and *pos2* each have the form *m.n* optionally followed by one or more of the flags **bdfinr**. A starting position specified by *+m.n* is interpreted to mean the *n*+1st character in the *m*+1st field. A missing *.n* means .0, indicating the first character of the *m*+1st field. If the **b** flag is in effect *n* is counted from the first non-blank in the *m*+1st field; *+m*.0**b** refers to the first non-blank character in the *m*+1st field.

A last position specified by *–m.n* is interpreted to mean the *n*th character (including separators) after the last character of the *m th* field. A missing *.n* means .0, indicating the last character of the *m*th field. If the **b** flag is in effect *n* is counted from the last leading blank in the *m*+1st field; *–m*.1**b** refers to the first non-blank in the *m*+1st field.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

EXAMPLES

Sort the contents of *infile* with the second field as the sort key:

sort +1 –2 infile

Sort, in reverse order, the contents of *infile1* and *infile2*, placing the output in *outfile* and using the first character of the second field as the sort key:

sort –r –o outfile +1.0 –1.2 infile1 infile2

Sort, in reverse order, the contents of *infile1* and *infile2* using the first non-blank character of the second field as the sort key:

sort –r +1.0b –1.1b infile1 infile2

Print the password file (*passwd*(4)) sorted by the numeric user ID (the third colon-separated field):

sort –t: +2n –3 /etc/passwd

Print the lines of the already sorted file *infile*, suppressing all but the first occurrence of lines having the same third field (the options **–um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

sort –um +2 –3 infile

FILES

/usr/tmp/stm???

SEE ALSO

comm(1), join(1), uniq(1)

WARNINGS

Comments and exits with non-zero status for various trouble conditions (for example, when input lines are too long), and for disorder discovered under the **–c** option. When the last line of an input file is missing a new-line character, *sort* appends one, prints a warning message, and continues.

sort does not guarantee preservation of relative line ordering on equal keys.

NAME

spell, hashmake, spellin, hashcheck – find spelling errors

SYNOPSIS

spell [**–v**] [**–b**] [**–x**] [**–l**] [+*local_file*] [*files*]

/usr/lib/spell/hashmake

/usr/lib/spell/spellin n

/usr/lib/spell/hashcheck *spelling_list*

DESCRIPTION

spell collects words from the named *files* and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no *files* are named, words are collected from the standard input.

spell ignores most *troff*(1), *tbl*(1), and *eqn*(1) constructions.

Under the **–v** option, all words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.

Under the **–b** option, British spelling is checked. Besides preferring *centre*, *colour*, *programme*, *speciality*, *travelled*, this option insists upon *-ise* in words like *standardise*, Fowler and the OED to the contrary notwithstanding.

Under the **–x** option, every plausible stem is printed with = for each word.

By default, *spell* (like *deroff*(1)) follows chains of included files (**.so** and **.nx** *troff*(1) requests), *unless* the names of such included files begin with **/usr/lib**. Under the **–l** option, *spell* will follow the chains of *all* included files.

Under the +*local_file* option, words found in *local_file* are removed from *spell*'s output. *local_file* is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to *spell*'s own spelling list) for each job.

The spelling list is based on many sources, and while more haphazard 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 light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings (see FILES). Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g., thier=thy-y+ier) that would otherwise pass.

Three routines help maintain and check the hash lists used by *spell*:

hashmake
Reads a list of words from the standard input and writes the corresponding nine-digit hash code on the standard output.

spellin
Reads *n* hash codes from the standard input and writes a compressed spelling list on the standard output.

hashcheck
Reads a compressed *spelling_list* and recreates the nine-digit hash codes for all the words in it; it writes these codes on the standard output.

FILES

D_SPELL=/usr/lib/spell/hlist[ab]	hashed spelling lists, American & British
S_SPELL=/usr/lib/spell/hstop	hashed stop list
H_SPELL=/usr/lib/spell/spellhist	history file
/usr/lib/spell/spellprog	program

SEE ALSO

deroff(1), sed(1), sort(1), tee(1)

BUGS

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions; typically, these are kept in a separate local file that is added to the hashed *spelling_list* via *spellin*.

With continued use, the history file will grow unchecked.

NAME

split – split a file into pieces

SYNOPSIS

split [*–n*] [*file* [*name*]]

DESCRIPTION

split reads *file* and writes it in *n*-line pieces (default 1000 lines) onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically, up to **zz** (a maximum of 676 files). *Name* cannot be longer than 12 characters. If no output name is given, **x** is default.

If no input file is given, or if **–** is given in its stead, then the standard input file is used.

SEE ALSO

bfs(1), csplit(1)

NAME

starter – information about the system for beginning users

SYNOPSIS

[**help**] **starter**

DESCRIPTION

The Help Facility command *starter* provides five categories of information about the system to assist new users.

The five categories are:

- commands a new user should learn first
- documents important for beginners
- education centers offering courses
- local environment information
- online teaching aids

The user may choose one of the above categories by entering its corresponding letter (given in the menu), or may exit to the shell by typing q (for "quit"). When a category is chosen, the user will receive one or more pages of information pertaining to it.

From any screen in the Help Facility, a user may execute a command via the shell (*sh*(1)) by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable SCROLL must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your **.profile** file (see **profile**(4)): "export SCROLL; SCROLL=no". If you later decide that scrolling is desired, SCROLL must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

SEE ALSO

glossary(1), help(1), locate(1), sh(1), usage(1).
term(5) in the *Programmer's Reference Manual.*

WARNINGS

If the shell variable TERM (see *sh*(1)) is not set in the user's **.profile** file, then TERM will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

strip – strip symbol and line number information from a common object file

SYNOPSIS

strip [**–l**] [**–x**] [**–b**] [**–r**] [**–V**] *filename* ...

DESCRIPTION

The *strip* command strips the symbol table and line number information from common object files, including archives. Once this has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

–l
Strip line number information only; do not strip any symbol table information.

–x
Do not strip static or external symbol information.

–b
Same as the **–x** option, but also do not strip scoping information (e.g., beginning and end of block delimiters).

–r
Do not strip static or external symbol information, or relocation information.

–V
Print the version of the strip command executing on the standard error output.

If there are any relocation entries in the object file and any symbol table information is to be stripped, *strip* will complain and terminate without stripping *filename* unless the **–r** option is used.

If the *strip* command is executed on a common archive file (see *ar*(4)) the archive symbol table will be removed. The archive symbol table must be restored by executing the *ar*(1) command with the **s** option before the archive can be link-edited by the *ld*(1) command. *strip* will produce appropriate warning messages when this situation arises.

strip is used to reduce the file storage overhead taken by the object file.

FILES

TMPDIR/strp* temporary files

TMPDIR is usually **/usr/tmp** but can be redefined by setting the environment variable TMPDIR (see *tempnam*() in *tmpnam*(3S)).

SEE ALSO

ar(1), as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4).

DIAGNOSTICS

strip: name: cannot open
if *name* cannot be read.

strip: name: bad magic
if *name* is not an appropriate common object file.

strip: name: relocation entries present; cannot strip
if *name* contains relocation entries and the **–r** flag is not used, the symbol table information cannot be stripped.

NAME

stty – set the options for a terminal

SYNOPSIS

stty [**–a**] [**–g**] [*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 (e.g., "^h" is CTRL-h; in this case, recall that CTRL-h is normally the same as the BACKSPACE key.) The sequence "^^" means that an option has a NULL value. For example, normally *stty* **–a** will report that the value of **swtch** is "^^"; however, if *shl (1)* has been invoked, **swtch** will have the value "^z".

–a
reports all the option settings.

–g
reports current settings in a form that can be used as an argument to another *stty* command.

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. The options are selected from the following:

Control Modes

parenb (**–parenb**) enable (disable) parity generation and detection.

parodd (**–parodd**) select odd (even) parity.

cs5 cs6 cs7 cs8 select character size (see *termio*(7)).

0 hang up phone line immediately.

110 300 600 1200 1800 2400 4800 9600 19200 38400
Set terminal baud rate to the number given, if possible. (Not all speeds are supported by all hardware interfaces.)

hupcl (**–hupcl**) hang up (do not hang up) Dataphone 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 (see *termio*(7)).

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 ASCII DC3 and started by sending an ASCII DC1.

ixany (–ixany) allow any character (only DC1) to restart output.

ixoff (–ixoff) request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

Output Modes

opost (–opost) post-process output (do not post-process output; ignore all other output modes).

olcuc (–olcuc) map (do not map) lowercase 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 (see *termio*(7)).
nl0 nl1	select style of delay for line-feeds (see *termio*(7)).
tab0 tab1 tab2 tab3	select style of delay for horizontal tabs (see *termio*(7)).
bs0 bs1	select style of delay for backspaces (see *termio*(7)).
ff0 ff1	select style of delay for form-feeds (see *termio*(7)).
vt0 vt1	select style of delay for vertical tabs (see *termio*(7)).

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).
xcase (–xcase)	canonical (unprocessed) upper/lowercase 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. Note that 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.
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.

stflush (–stflush) enable (disable) flush on a synchronous line after every *write*(2).

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* is **erase, kill, intr, quit, swtch, eof, ctab, min,** or **time** (**ctab** is used with **–stappl**; **min** and **time** are used with **–icanon**; see *termio*(7)). If *c* is preceded by a caret (^) (escaped from the shell), then the value used is the corresponding CTRL character (e.g., "^**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**.

oddp enable **parenb, cs7,** and **parodd**.

–parity, –evenp, or **–oddp** disable **parenb**, 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.**

SEE ALSO

tabs(1).
ioctl(2) in the *Programmer's Reference Manual.*
termio(7) in the *System Administrator's Reference Manual.*

NAME

sum – print checksum and block count of a file

SYNOPSIS

sum [**–r**] *file*

DESCRIPTION

sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. It is typically used to look for bad spots, or to validate a file communicated over some transmission line. The option **–r** causes an alternate algorithm to be used in computing the checksum.

SEE ALSO

wc(1).

DIAGNOSTICS

"Read error" is indistinguishable from end of file on most devices; check the block count.

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [*tabspec*] [**–T***type*] [**+m***n*]

DESCRIPTION

tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user's terminal must have remotely-settable hardware tabs.

tabspec

Four types of tab specification are accepted for *tabspec*. They are described below: canned (*–code*), repetitive (*–n*), arbitrary (*n1,n2,...*), and file (*—file*). If no *tabspec* is given, the default value is –8, i.e., "standard" tabs. The lowest column number is 1. Note that for *tabs*, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0, e.g., the DASI 300, DASI 300s, and DASI 450.

–code

Use one of the codes listed below to select a *canned* set of tabs. The legal codes and their meanings are:

–a
1,10,16,36,72
Assembler, IBM S/370, first format

–a2
1,10,16,40,72
Assembler, IBM S/370, second format

–c
1,8,12,16,20,55
COBOL, normal format

–c2
1,6,10,14,49
COBOL compact format (columns 1-6 omitted). Using this code, the first typed character corresponds to card column 7, one space gets 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–c2 m6 s66 d:>

–c3 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1-6 omitted), with more tabs than **–c2**. This is the recommended format for COBOL. The appropriate format specification is (see *fspec*(4)):

<:t–c3 m6 s66 d:>

–f 1,7,11,15,19,23
FORTRAN

–p 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

–s 1,10,55
SNOBOL

–u 1,12,20,44
UNIVAC 1100 Assembler

–n
A *repetitive* specification requests tabs at columns 1+*n*, 1+2**n*, etc. Of particular importance is the value **8**: this represents the "standard" tab setting, and is the most likely tab setting to be found at a terminal. Another special case is the value **0**, implying no tabs at all.

n1,n2,...
The *arbitrary* format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the formats **1,10,20,30**, and **1,10,+10,+10** are considered identical.

—file
If the name of a *file* is given, *tabs* reads the first line of the file, searching for a format specification (see *fspec*(4)). If it finds one there, it sets the tab stops according to it, otherwise it sets them as –8. This type of specification may be used to make sure that a tabbed file is printed with correct tab settings, and would be used with the *pr*(1) command:

tabs — file; pr file

Any of the following also may be used; if a given flag occurs more than once, the last value given takes effect:

–T*type*
tabs usually needs to know the type of terminal in order to set tabs and always needs to know the type to set margins. *type* is a name listed in *term*(5). If no **–T** flag is supplied, *tabs* uses the value of the environment variable **TERM**. If **TERM** is not defined in the *environment* (see *environ*(5)), *tabs* tries a sequence that will work for many terminals.

+m*n*
The margin argument may be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n+1* the left margin. If **+m** is given without a value of *n*, the value assumed is **10**. For a TermiNet, the first value in the tab list should be **1**, or the margin will move even further to the right. The normal (leftmost) margin on most terminals is obtained by **+m0**. The margin for most terminals is reset only when the **+m** flag is given explicitly.

Tab and margin setting is performed via the standard output.

EXAMPLES

tabs –a
example using *–code* (*canned* specification) to set tabs to the settings required by the IBM assembler: columns 1, 10, 16, 36, 72.

tabs –8
example of using *–n* (*repetitive* specification), where *n* is **8**, causes tabs to be set every eighth position:
1+(1*8), 1+(2*8), ... which evaluate to columns 9, 17, ...

tabs 1,8,36
example of using *n1,n2,...* (*arbitrary* specification) to set tabs at columns 1, 8, and 36.

tabs —$HOME/fspec.list/att4425
example of using *—file* (*file* specification) to indicate that tabs should be set according to the first line of **$HOME/fspec.list/att4425** (see *fspec*(4)).

DIAGNOSTICS

illegal tabs
when arbitrary tabs are ordered incorrectly

illegal increment
when a zero or missing increment is found in an arbitrary specification

unknown tab code
when a *canned* code cannot be found

can't open
if *—file* option used, and file can't be opened

file indirection
if *—file* option used and the specification in that file points to yet another file. Indirection of this form is not permitted

SEE ALSO

newform(1), pr(1), tput(1)
fspec(4), terminfo(4), environ(5), term(5) in the *Programmer's Reference Manual.*

NOTE

There is no consistency among different terminals regarding ways of clearing tabs and setting the left margin.

tabs clears only 20 tabs (on terminals requiring a long sequence), but is willing to set 64.

WARNING

The *tabspec* used with the *tabs* command is different from the one used with the *newform*(1) command. For example, **tabs –8** sets every eighth position; whereas **newform –i–8** indicates that tabs are set every eighth position.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [± [*number*][**lbc** [**f**]]] [*file*]

DESCRIPTION

tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning, or –*number* from the end of the input (if *number* is null, the value 10 is assumed). *Number* is counted in units of lines, blocks, or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

With the **–f** ("follow") option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop, wherein it sleeps for a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process. For example, the command:

tail –f fred

will print the last ten lines of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed. As another example, the command:

tail –15cf fred

will print the last 15 characters of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed.

SEE ALSO

dd(1M).

BUGS

Tails relative to the end of the file are stored in a buffer, thus, are limited in length. Various kinds of anomalous behavior may happen with character special files.

WARNING

The *tail* command will only tail the last 4096 bytes of a file regardless of its line count.

NAME

tar – tape file archiver

SYNOPSIS

/etc/tar –c[iLvwfb[#*s*]] *device block files ...*
/etc/tar –r[iLvwb[#*s*]] *device block* [*files ...*]
/etc/tar –t[iLvf[#*s*] *device*
/etc/tar –u[iLvwb[#*s*]] *device block* [*files ...*]
/etc/tar –x[iLlmovwf[#*s*]] *device* [*files ...*]

DESCRIPTION

tar saves and restores files on magnetic tape. Its actions are controlled by the *key* argument. The *key* is a string of characters containing one function letter (**c, r, t, u,** or **x**) and possibly followed by one or more function modifiers (**b, f, i,** L, **v, w,** and **#**). Other arguments to the command are *files* (or directory names) specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r
Replace. The named *files* are written on the end of the tape. The **c** function implies this function.

x
Extract. The named *files* are extracted from the tape. If a named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. Use the file or directory's relative path when appropriate, or *tar* will not find a match. The owner, modification time, and mode are restored (if possible). If no *files* argument is given, the entire content of the tape is extracted. Note that if several files with the same name are on the tape, the last one overwrites all earlier ones.

t
Table. The names and other information for the specified files are listed each time that they occur on the tape. The listing is similar to the format produced by the **ls -l** command. If no *files* argument is given, all names on the tape are listed.

u
Update. The named *files* are added to the tape if they are not already there, or have been modified since last written on that tape. This key implies the **r** key.

c

Create a new tape; writing begins at the beginning of the tape, instead of after the last file. This key implies the **r** key.

The following characters may be used in addition to the letter that selects the desired function. Use them in the order shown in the synopsis:

i

This modifier causes *tar* to ignore symbolic links.

L

This modifier causes *tar* to follow symbolic links. The default is not to follow links. If an archive is made from a tree containing symbolic links, it will record the path associated with each link. When it is restored, the symbolic links will be re-made. If **–L** is specified, the actual file pointed to by the link is archived instead of the symbolic link contents.

#s

This modifier determines the drive on which the tape is mounted (replace **#** with the drive number) and the speed of the drive (replace **s** with **l**, **m**, or **h** for low, medium or high). The modifier tells *tar* to use a drive other than the default drive, or the drive specified with the -f option. For example, with the **5h** modifier, *tar* would use **/dev/mt/5h** or **/dev/mt0** instead of the default drives **/dev/mt/0m** or **/dev/mt0**, respectively. However, if for example, "**-f /dev/rmt0 5h**" appeared on the command line, *tar* would use **/dev/rmt5h** or **/devmt0**. The default entry is **0m**.

v

Verbose. Normally, *tar* does its work silently. The **v** (verbose) option causes it to type the name of each file it treats, preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.

w

What. This causes *tar* to print the action to be taken, followed by the name of the file, and then wait for the user's confirmation. If a word beginning with **y** is given, the action is performed. Any other input means "no". This is not valid with the **t** key.

f

File. This causes *tar* to use the *device* argument as the name of the archive instead of **/dev/rmt/ctape**. (Note that the cartridge tape in SYSTEM V/88 is referenced as **/dev/rmt/ctape**.) If the name of the file

is –, *tar* writes to the standard output or reads from the standard input, whichever is appropriate. Thus, *tar* can be used as the head or tail of a pipeline. *tar* can also be used to move hierarchies with the command:

cd fromdir; tar cf – . | (cd todir; tar xf –)

b

Blocking Factor. This causes *tar* to use the *block* argument as the blocking factor for tape records. The default is 1, the maximum is 20. This function should not be supplied when operating on regular archives or block special devices. It is mandatory however, when reading archives on raw magnetic tape archives (see **f** above). The block size is determined automatically when reading tapes created on block special devices (key letters **x** and **t**).

l

Link. This tells *tar* to complain if it cannot resolve all of the links to the files being dumped. If **l** is not specified, no error messages are printed.

m

Modify. This tells *tar* to not restore the modification times. The modification time of the file will be the time of extraction.

o

Ownership. This causes extracted files to take on the user and group identifier of the user running the program, instead of those on tape. This is only valid with the **x** key.

FILES

/dev/mt/*
/tmp/tar*
/dev/rmt/*

SEE ALSO

ar(1), cpio(1), ls(1).

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if enough memory is not available to hold the link tables.

BUGS

There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow. The **u** option will not work on 88K cartridge tape devices.
The **b** option should not be used with archives that are going to be updated. The current magnetic tape driver cannot backspace raw magnetic tape. If the archive is on a disk file, the **b** option should not be used at all, because updating an archive stored on disk can destroy it.
The current limit on file name length is 100 characters.
tar does not copy empty directories or special files.

NAME

tee – pipe fitting

SYNOPSIS

tee [**–i**] [**–a**] [*file*] ...

DESCRIPTION

tee transcribes the standard input to the standard output and makes copies in the *files*. The

–i ignore interrupts;

–a causes the output to be appended to the *files* rather than overwriting them.

NAME

test – condition evaluation command

SYNOPSIS

test *expr*
[*expr*]

DESCRIPTION

test evaluates the expression *expr* and, if its value is true, sets a zero (true) exit status; otherwise, sets a non-zero (false) exit status. *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.

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.

–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.

–l *file*
true if *file* exists and is a symbolic link.

–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**.

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, therefore, must be quoted.

SEE ALSO

find(1), sh(1).

WARNING

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 superuser.

The = and != operators have a higher precedence than the **–r** through **–n** operators, and = and != always expect arguments; therefore, they 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 an **–a** or an **–o** is the second argument.

NAME

time – time a command

SYNOPSIS

time *command*

DESCRIPTION

The *command* is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The times are printed on standard error.

SEE ALSO

times(2) in the *Programmer's Reference Manual.*

NAME

timex – time a command; report process data and system activity

SYNOPSIS

timex [*options*] *command*

DESCRIPTION

The given *command* is executed; the elapsed time, user time and system time spent in execution are reported in seconds. Optionally, process accounting data for the *command* and all its children can be listed or summarized, and total system activity during the execution interval can be reported.

The output of *timex* is written on standard error.

The *options* are:

–p
List process accounting records for *command* and all its children. This option works only if the process accounting software is installed. Suboptions **f , h , k , m , r ,** and **t** modify the data items reported. The options are:

–f Print the *fork/exec* flag and system exit status columns in the output.

–h Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution. This "hog factor" is computed as:

(total CPU time)/(elapsed time).

–k Instead of memory size, show total kcore-minutes.

–m Show mean core size (the default).

–r Show CPU factor (user time/(system-time + user-time).

–t Show separate system and user CPU times. The number of blocks read or written and the number of characters transferred are always reported.

–o
Report the total number of blocks read or written and total characters transferred by *command* and all its children. This option works only if the process accounting software is installed.

–s

Report total system activity (not just that due to *command*) that occurred during the execution interval of *command*. All the data items listed in *sar*(1) are reported.

SEE ALSO

sar(1)

WARNING

Process records associated with *command* are selected from the accounting file **/usr/adm/pacct** by inference, since process genealogy is not available. Background processes having the same user-id, terminal-id, and execution time window will be spuriously included.

EXAMPLES

A simple example:

timex –ops sleep 60

A terminal session of arbitrary complexity can be measured by timing a sub-shell:

timex –opskmt sh

session commands
EOT

NAME

touch – update access and modification times of a file

SYNOPSIS

touch [**–amc**] [*mmddhhmm* [*yy*]] *files*

DESCRIPTION

touch causes the access and modification times of each argument to be updated. The file name is created if it does not exist. If no time is specified (see *date*(1)) the current time is used. The **–a** and **–m** options cause touch to update only the access or modification times respectively (default is **–am**). The **–c** option silently prevents *touch* from creating the file if it did not previously exist.

The return code from *touch* is the number of files for which the times could not be successfully modified (including files that did not exist and were not created).

SEE ALSO

date(1).
utime(2) in the *Programmer's Reference Manual*.

NAME

tput – initialize a terminal or query terminfo database

SYNOPSIS

tput [**–T** *type*] *capname* [*parms* ...]

tput [**–T** *type*] **init**

tput [**–T** *type*] **reset**

tput [**–T** *type*] **longname**

tput –S << *file*

DESCRIPTION

tput uses the *terminfo*(4) database to make the values of terminal-dependent capabilities and information available to the shell (see *sh*(1)), to initialize or reset the terminal, or return the long name of the requested terminal type.

tput outputs a string if the attribute (*cap*ability *name*) is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, *tput* simply sets the exit code (**0** for TRUE if the terminal has the capability, **1** for FALSE if it does not), and produces no output. Before using a value returned on standard output, the user should test the exit code ($?, see *sh*(1)) to be sure it is **0**. (See EXIT CODES and DIAGNOSTICS.) For a complete list of capabilities and the *capname* associated with each, see *terminfo*(4).

–T*type*
indicates the *type* of terminal. Normally, this option is unnecessary because the default is taken from the environment variable TERM. If **–T** is specified, the shell variables LINES and COLUMNS and the layer size (see *layers*(1)) will not be referenced.

capname
indicates the attribute from the *terminfo*(4) database.

parms
If the attribute is a string that takes parameters, the arguments *parms* will be instantiated into the string. An all numeric argument will be passed to the attribute as a number.

–S

allows more than one capability per invocation of *tput*. The capabilities must be passed to *tput* from the standard input instead of from the command line (see example). Only one *capname* is allowed per line. The **–S** option changes the meaning of the **0** and **1** boolean and string exit codes (see **EXIT CODES**).

init

If the *terminfo*(4) database is present and an entry for the user's terminal exists (see **–T***type*, above), the following will occur: (1) if present, the terminal's initialization strings will be output (**is1**, **is2**, **is3**, **if**, **iprog**), (2) any delays (e.g., newline) specified in the entry will be set in the TTY driver, (3) tabs expansion will be turned on or off according to the specification in the entry, and (4) if tabs are not expanded, standard tabs will be set (every 8 spaces). If an entry does not contain the information needed for any of the four above activities, that activity will silently be skipped.

reset

Instead of putting out initialization strings, the terminal's reset strings will be output if present (**rs1**, **rs2**, **rs3**, **rf**). If the reset strings are not present, but initialization strings are, the initialization strings will be output. Otherwise, **reset** acts identically to **init**.

longname

If the *terminfo*(4) database is present and an entry for the user's terminal exists (see **–T***type* above), then the long name of the terminal will be put out. The long name is the last name in the first line of the terminal's description in the *terminfo*(4) database (see *term*(5)).

EXAMPLES

tput init

Initialize the terminal according to the type of terminal in the environmental variable TERM. This command should be included in everyone's **.profile** after the environmental variable TERM has been exported, as illustrated on the *profile*(4) manual page.

tput –T5620 reset

Reset an AT&T 5620 terminal, overriding the type of terminal in the environmental variable TERM.

tput cup 0 0

Send the sequence to move the cursor to row **0**, column **0** (upper left corner of the screen, usually known as "home" cursor position).

tput clear
Echo the clear-screen sequence for the current terminal.

tput cols
Print the number of columns for the current terminal.

tput -T450 cols
Print the number of columns for the 450 terminal.

bold=‘tput smso‘
offbold=‘tput rmso‘
Set the shell variables **bold**, to begin stand-out mode sequence, and **offbold**, to end standout mode sequence, for the current terminal. This might be followed by a prompt:

echo "${bold}Please type in your name: ${offbold}\c"

tput hc
Set exit code to indicate if the current terminal is a hardcopy terminal.

tput cup 23 4
Send the sequence to move the cursor to row 23, column 4.

tput longname
Print the long name from the *terminfo*(4) database for the type of terminal specified in the environmental variable TERM.

tput -S <<!
> clear
> cup 10 10
> bold
> !
This example shows *tput* processing several capabilities in one invocation. This example clears the screen, moves the cursor to position 10, 10 and turns on bold (extra bright) mode. The list is terminated by an exclamation mark (!) on a line by itself.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/include/curses.h	*curses*(3X) header file
/usr/include/term.h	*terminfo*(4) header file
/usr/lib/tabset/*	tab settings for some terminals in a format appropriate to be output to the terminal (escape sequences that set margins and tabs); for more information, see the *Tabs and Initialization* section of *terminfo*(4).

SEE ALSO

stty (1), tabs (1)
profile(4), terminfo(4) in the *System Administrator's Reference Manual.*
Chapter 10 of the *Programmer's Guide.*

EXIT CODES

If *capname* is of type boolean, a value of **0** is set for TRUE and **1** for FALSE unless the **–S** option is used.

If *capname* is of type string, a value of **0** is set if the *capname* is defined for this terminal *type* (the value of *capname* is returned on standard output); a value of **1** is set if *capname* is not defined for this terminal *type* (a NULL value is returned on standard output).

If *capname* is of type boolean or string and the **–S** option is used, a value of **0** is returned to indicate that all lines were successful. No indication of which line failed can be given so exit code **1** will never appear. Exit codes **2, 3,** and **4** retain their usual interpretation.

If *capname* is of type integer, a value of **0** is always set, whether or not *capname* is defined for this terminal *type*. To determine if *capname* is defined for this terminal *type*, the user must test the value of standard output. A value of **–1** means that *capname* is not defined for this terminal *type*.

Any other exit code indicates an error; see DIAGNOSTICS.

DIAGNOSTICS

tput prints the following error messages and sets the corresponding exit codes.

Exit Code	Error Message
0	–1 (*capname* is a numeric variable that is not specified in the *terminfo*(4) database for this terminal type, for example: **tput –T450 lines** and **tput –T2621 xmc**)
1	no error message is printed, see **EXIT CODES**, above.
2	usage error
3	unknown terminal *type* or no *terminfo*(4) database
4	unknown *terminfo*(4) capability *capname*

NAME

tr – translate characters

SYNOPSIS

tr [**–cds**] [*string1* [*string2*]]

DESCRIPTION

tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. Any combination of the options **–cds** may be used:

–c
Complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 001 through 377 octal.

–d
Deletes all input characters in *string1*.

–s
Squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[a–z]
Stands for the string of characters whose ASCII codes run from character **a** to character **z**, inclusive.

[a*n]
Stands for *n* repetitions of **a**. If the first digit of *n* is **0**, *n* is considered octal; otherwise, *n* is taken to be decimal. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character \ may be used as in the shell to remove special meaning from any character in a string. In addition, \ followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

EXAMPLE

The following example creates a list of all the words in *file1* one per line in *file2*, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the shell; 012 is the ASCII code for newline.

```
tr –cs "[A–Z][a–z]" "[\012*]" <file1 >file2
```

SEE ALSO

ed(1), sh(1).
ascii(5) in the *Programmer's Reference Manual.*

BUGS

Will not handle ASCII NUL in *string1* or *string2*; always deletes NUL from input.

NAME

true, false – provide truth values

SYNOPSIS

true

false

DESCRIPTION

true does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1), for example:

```
while true
do
  command
done
```

SEE ALSO

sh(1).

DIAGNOSTICS

true has exit status zero, *false* nonzero.

NAME

tsort – topological sort

SYNOPSIS

tsort [*file*]

DESCRIPTION

The *tsort* command produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1).

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

NAME

tt — convert and copy a file

SYNOPSIS

tt if=*file* **of**=*file* **bs**=*n* **count**=*n*

DESCRIPTION

tt copies the specified input file to the specified output; the raw devices will be used by default:

if=*file*	input disk alias searched for in the **permissions** file.
of=*file*	output disk alias searched for in the **permissions** file.
bs=*n*	set both input and output block size.
count=*n*	copy only *n* input records.

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

FILES

/bin/dd
/etc/filesys **permissions** file

SEE ALSO

cp(1), dd(1) in the *User's Reference Manual.*
filesys(4) in the *Programmer's Reference Manual.*

DIAGNOSTICS

After completion, *tt* reports the number of whole and partial input and output blocks:

full+partial records in
full+partial records out

BUGS

Only files listed in the **permissions** file may be accessed.

NAME

tty – get the name of the terminal

SYNOPSIS

tty [–l] [–s]

DESCRIPTION

tty prints the pathname of the user's terminal:

–l
prints the synchronous line number to which the user's terminal is connected if it is on an active synchronous line.

–s
inhibits printing of the terminal pathname, allowing you to test just the exit code.

Exit Codes	Description
2	if invalid options were specified
0	if standard input is a terminal
1	otherwise

DIAGNOSTICS

"not on an active synchronous line" if the standard input is not a synchronous terminal and **–l** is specified.

"not a tty" if the standard input is not a terminal and **–s** is not specified.

NAME

umask – set file-creation mode mask

SYNOPSIS

umask [ooo]

DESCRIPTION

The user file-creation mode mask is set to *ooo*. The three octal digits refer to read/write/execute permissions for *owner*, *group*, and *others*, respectively (see *chmod*(2) and *umask*(2)). The value of each specified digit is subtracted from the corresponding "digit" specified by the system for the creation of a file (see *creat*(2)). For example, **umask 022** removes *group* and *others* write permission (files normally created with mode **777** become mode **755**; files created with mode **666** become mode **644**).

If *ooo* is omitted, the current value of the mask is printed.

umask is recognized and executed by the shell.

umask can be included in the user's **.profile** (see *profile*(4)) and invoked at login to automatically set the user's permissions on files or directories created.

SEE ALSO

chmod(1), sh(1).
chmod(2), creat(2), umask(2), profile(4) in the *Programmer's Reference Manual*.

NAME

uname – print name of current system

SYNOPSIS

uname [**–snrvma**]
uname [**–S** *system name*]

DESCRIPTION

uname prints the name of the current system on standard output. It is mainly useful to determine which system one is using. The options cause selected information returned by *uname*(2) to be printed:

–s prints the system name (default).

–n prints the node name. (The node name is the name by which the system is known to a communications network.)

–r prints the operating system release.

–v prints the operating system version.

–m prints the machine hardware name.

–a prints all the above information.

The system name and the node name may be changed by specifying a system-name argument to the **–S** option. The system-name argument is restricted to 8 characters. Only the superuser is allowed this capability.

SEE ALSO

uname(2) in the *Programmer's Reference Manual.*

NAME

unget – undo a previous get of an SCCS file

SYNOPSIS

unget [*–r SID*] [**–s**] [**–n**] *files*

DESCRIPTION

unget undoes the effect of a **get –e** done before creating the intended new delta. If a directory is named, *unget* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Keyletter arguments apply independently to each named file.

–r*SID*

Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the "new delta"). The use of this keyletter is necessary only if two or more outstanding *get*s for editing on the same SCCS file were done by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

–s

Suppresses the printout, on the standard output, of the intended delta's *SID*.

–n

Causes the retention of the gotten file which would normally be removed from the current directory.

SEE ALSO

delta(1), get(1), sact(1)
help(1) in the *User's Reference Manual*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

uniq – report repeated lines in a file

SYNOPSIS

uniq [**–udc** [*+n*] [*–n*]] [*input* [*output*]]

DESCRIPTION

uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. *Input* and *output* should always be different. Note that repeated lines must be adjacent in order to be found; see *sort*(1). If the **–u** flag is used, just the lines that are not repeated in the original file are output. The **–d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **–u** and **–d** mode outputs.

The **–c** option supersedes **–u** and **–d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

–n
: The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+n
: The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

comm(1), sort(1).

NAME

units – conversion program

SYNOPSIS

units

DESCRIPTION

units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
        * 2.540000e+00
        / 3.937008e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have: 15 lbs force/in2
You want: atm
        * 1.020689e+00
        / 9.797299e-01
```

units only does multiplicative scale changes; thus, it can convert Kelvin to Rankine, but not Celsius to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter,
c	speed of light,
e	charge on an electron,
g	acceleration of gravity,
force	same as **g**,
mole	Avogadro's number,
water	pressure head per unit height of water,
au	astronomical unit.

pound is not recognized as a unit of mass; **lb** is. Compound names are run together, (e.g., **lightyear**). British units that differ from their U.S. counterparts are prefixed: **brgallon**. For a complete list of units, type:

cat /usr/lib/unittab

FILES

/usr/lib/unittab

NAME

usage – retrieve a command description and usage examples

SYNOPSIS

[help] usage [–d] [–e] [–o] [*command_name* **]**

DESCRIPTION

The Help Facility command *usage* retrieves information about commands. With no argument, *usage* displays a menu screen prompting the user for the name of a command, or allows the user to retrieve a list of commands supported by *usage*. The user may also exit to the shell by typing **q** (for "quit).

After a command is selected, the user is asked to choose among a description of the command, examples of typical usage of the command, or descriptions of the command's options. Then, based on the user's request, the appropriate information will be printed.

A command name may also be entered at shell level as an argument to *usage*. To receive information on the command's description, examples, or options, the user may use the **–d**, **–e**, or **–o** options respectively. (The default option is **–d**.)

From any screen in the Help Facility, a user may execute a command via the shell (*sh*(1)) by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable SCROLL must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your **.profile** file (see *profile*(4)): "export SCROLL ; SCROLL=no". If you later decide that scrolling is desired, SCROLL must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

SEE ALSO

glossary(1), help(1), locate(1), sh(1), starter(1).
term(5) in the *Programmer's Reference Manual*.

WARNINGS

If the shell variable TERM (see *sh*(1)) is not set in the user's **.profile** file, then TERM will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

val – validate SCCS file

SYNOPSIS

val –
val [**–s**] [**–r***SID*] [**–m***name*] [**–y***type*] *files*

DESCRIPTION

val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of keyletter arguments, which begin with a **–**, and named files.

val has a special argument, **–**, which causes reading of the standard input until an EOF condition is detected. Each line read is independently processed as if it were a command line argument list.

val generates diagnostic messages on the standard output for each command line and file processed, and also returns a single 8-bit code upon exit as described below.

The following defines keyletter arguments; the effects of any keyletter argument apply independently to each named file on the command line:

–s
The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

–r*SID*
The argument value *SID* is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., r1 is ambiguous because it physically does not exist but implies 1.1, 1.2, which may exist) or invalid (e. g., r1.0 or r1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

–mname
The argument value *name* is compared with the s-1SCCS %M% keyword in *file*.

–ytype
The argument value *type* is compared with the SCCS %Y% keyword in *file*.

The 8-bit code returned by *val* is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;
bit 2 = corrupted SCCS file;
bit 3 = cannot open file or file not SCCS;
bit 4 = *SID* is invalid or ambiguous;
bit 5 = *SID* does not exist;
bit 6 = %Y%, **–y** mismatch;
bit 7 = %M%, **–m** mismatch;

Note that *val* can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned —a logical OR of the codes generated for each command line and file processed.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

val can process up to 50 files on a single command line. Any number above 50 will produce a **core** dump.

NAME

vc – version control

SYNOPSIS

vc [**–a**] [**–t**] [**–c***char*] [**–s**] [*keyword=value ... keyword=value*]

DESCRIPTION

The *vc* command copies lines from the standard input to the standard output under control of its *arguments* and *control statements* encountered in the standard input. In the process of performing the copy operation, user declared *keywords* may be replaced by their string *value* when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests (in control statements) of keyword values specified in control statements or as *vc* command arguments.

A control statement is a single line beginning with a control character, except as modified by the **–t** keyletter (see below). The default control character is colon (:), except as modified by the **–c** keyletter (see below). Input lines beginning with a backslash (\) followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or less alphanumerics; the first must be alphabetic. A value is any ASCII string that can be created with *ed*(1); a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Replacement of keywords by values is done whenever a keyword surrounded by control characters is encountered on a version control statement. The **–a** keyletter (see below) forces replacement of keywords in *all* lines of text. An uninterpreted control character may be included in a value by preceding it with ****. If a literal **** is desired, then it too must be preceded by ****.

Keyletter Arguments

–a

Forces replacement of keywords surrounded by control characters with their assigned value in *all* text lines and not just in *vc* statements.

–t
All characters from the beginning of a line up to and including the first *tab* character are ignored for the purpose of detecting a control statement. If one is found, all characters up to and including the *tab* are discarded.

–cchar
Specifies a control character to be used in place of **:**.

–s
Silences warning messages (not error) that are normally printed on the diagnostic output.

Version Control Statements

:dcl keyword[, ..., keyword]
Used to declare keywords. All keywords must be declared.

:asg keyword=value
Used to assign values to keywords. An **asg** statement overrides the assignment for the corresponding keyword on the *vc* command line and all previous **asg**'s for that keyword. Keywords declared, but not assigned values have **NULL** values.

:if condition
⋮
:end
Used to skip lines of the standard input. If the condition is true all lines between the *if* statement and the matching *end* statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control statements. Note that intervening *if* statements and matching *end* statements are recognized solely for the purpose of maintaining the proper *if-end* matching.

The syntax of a condition is:

```
<cond>      ::= [ "not" ] <or>
<or>        ::= <and> | <and> "|" <or>
<and>       ::= <exp> | <exp> "&" <and>
<exp>       ::= "(" <or> ")" | <value> <op> <value>
<op>        ::= "=" | "!=" | "<" | ">"
<value>     ::= <arbitrary ASCII string> | <numeric string>
```

The available operators and their meanings are:

=	equal
!=	not equal
&	and
|	or
>	greater than
<	less than
( )	used for logical groupings
not	may only occur immediately after the *if*, and when present, inverts the value of the entire condition

The > and < operate only on unsigned integer values (e.g., : 012 > 12 is false). All other operators take strings as arguments (e.g., : 012 != 12 is true). The precedence of the operators (from highest to lowest) is:

= != > < all of equal precedence
&
|

Parentheses may be used to alter the order of precedence.
Values must be separated from operators or parentheses by at least one blank or tab.

::text
Used for keyword replacement on lines that are copied to the standard output. The two leading control characters are removed; keywords surrounded by control characters in text are replaced by their value before the line is copied to the output file. This action is independent of the –a keyletter.

:on

:off
Turn on or off keyword replacement on all lines.

:ctl char
Change the control character to char.

:msg message
Prints the given message on the diagnostic output.

:err message
Prints the given message followed by:

ERROR: err statement on line ... (915)

on the diagnostic output. *vc* halts execution, and returns an exit code of 1.

SEE ALSO

ed(1), help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

EXIT CODES

0 normal
1 any error

NAME

vi – screen-oriented (visual) display editor based on ex

SYNOPSIS

vi [**–t** *tag*] [**–r** *file*] [**–L**] [**–w***n*] [**–R**] [**–x**] [**–C**]
[**–c** *command*] *file* ...
view [**–t** *tag*] [**–r** *file*] [**–L**] [**–w** *n*] [**–R**] [**–x**] [**–C**]
[**–c** *command*] *file* ...
vedit [**–t** *tag*] [**–r** *file*] [**–L**] [**–w** *in*] [**–R**] [**–x**] [**–C**]
[**–c** *command*] *file* ...

DESCRIPTION

The *vi* (visual) program is a display-oriented text editor based on an underlying line editor *ex*(1). It is possible to use the command mode of ex from within *vi* and vice-versa. The visual commands are described on this manual page; how to set options (like automatically numbering lines and automatically starting a new output line when you type carriage return) and all *ex*(1) line editor commands are described on the *ex*(1) manual page.

When using *vi*, changes you make to the file are reflected in what you see on your terminal screen. The position of the cursor on the screen indicates the position within the file.

Invocation Options

The following invocation options are interpreted by *vi* (previously documented options are discussed in the **NOTES** section):

–t *tag*
Edit the file containing the *tag* and position the editor at its definition.

–r *file*
Edit *file* after an editor or system crash. (Recovers the version of *file* that was in the buffer when the crash occurred.)

–L
List the name of all files saved as the result of an editor or system crash.

–w*n*
Set the default window size to *n*. This is useful when using the editor over a slow speed line.

–R
Readonly mode; the **readonly** flag is set, preventing accidental overwriting of the file.

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

–C
Encryption option; same as the **–x** option, except that *vi* simulates the C command of *ex*(1). The C command is like the *X* command of *ex*(1), except that all text read in is assumed to have been encrypted.

–c *command*
Begin editing by executing the specified editor *command* (usually a search or positioning command).

The *file* argument indicates one or more files to be edited.

The *view* invocation is the same as *vi* except that the **readonly** flag is set.

The *vedit* invocation is intended for beginners. It is the same as *vi* except that the **report** flag is set to 1, the **showmode** and **novice** flags are set, and **magic** is turned off. These defaults make it easier to learn how to use *vi*.

vi Modes

Command
Normal and initial mode. Other modes return to command mode upon completion. **ESC** (escape) is used to cancel a partial command.

Input
Entered by setting any of the following options: **a A i I o O c C s S R .** Arbitrary text may then be entered. Input mode is normally terminated with **ESC** character, or, abnormally, with an interrupt.

Last line
Reading input for **:** / **?** or **!**; terminate by typing a carriage return; an interrupt cancels termination.

COMMAND SUMMARY

In the descriptions, CR stands for carriage return and ESC stands for the escape key.

Sample commands

← ↓ ↑ →	arrow keys move the cursor
h j k l	same as arrow keys
i*text*ESC	insert *text*
cw*new*ESC	change word to *new*
ea*s*ESC	pluralize word (end of word; append s; escape from input state)
x	delete a character
dw	delete a word
dd	delete a line
3dd	delete 3 lines
u	undo previous change
ZZ	exit *vi*, saving changes
:q!CR	quit, discarding changes
/*text*CR	search for *text*
^U ^D	scroll up or down
:*cmd*CR	any *ex* or *ed* command

Counts before vi commands

Numbers may be typed as a prefix to some commands. They are interpreted in one of three ways:

line/column number	**z G |**
scroll amount	**^D ^U**
repeat effect	most of the rest

Interrupting, canceling

ESC	end insert or incomplete command
DEL	(delete or rubout) interrupts

File manipulation

ZZ	if file modified, write and exit; otherwise, exit
:wCR	write back changes
:w!CR	forced write, if permission originally not valid
:qCR	quit
:q!CR	quit, discard changes
:e *name*CR	edit file *name*
:e!CR	reedit, discard changes
:e + *name*CR	edit, starting at end

:e +*n***CR**	edit starting at line *n*
:e #CR	edit alternate file
:e! #CR	edit alternate file, discard changes
:w *name***CR**	write file *name*
:w! *name***CR**	overwrite file *name*
:sh CR	run shell, then return
:! *cmd***CR**	run *cmd*, then return
:n CR	edit next file in arglist
:n *args***CR**	specify new arglist
^G	show current file and line
:ta *tag***CR**	position cursor to *tag*

In general, any **ex** or **ed** command (such as *substitute* or *global*) may be typed, preceded by a colon and followed by a carriage return.

Positioning within file

^F	forward screen
^B	backward screen
^D	scroll down half screen
^U	scroll up half screen
*n***G**	go to the beginning of the specified line (end default), where *n* is a line number
/pat	next line matching *pat*
?*pat*	previous line matching *pat*
n	repeat last / or ? command
N	reverse last / or ? command
/pat/+n	nth line after *pat*
?*pat***?–***n*	nth line before *pat*
]]	next section/function
[[	previous section/function
(	beginning of sentence
)	end of sentence
{	beginning of paragraph
}	end of paragraph
%	find matching **( ) { }** or **}**

Adjusting the screen

^L	clear and redraw window
^R	clear and redraw window if **^L** is → key
zCR	redraw screen with current line at top of window
z–CR	redraw screen with current line at bottom of window
z .CR	redraw screen with current line at center of window

/*pat*/z–CR	move *pat* line to bottom of window
z*n*.CR	use *n*-line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and returning

``	move cursor to previous context
''	move cursor to first non-white space in line
m*x*	mark current position with the ASCII lowercase letter *x*
`*x*	move cursor to mark *x*
'*x*	move cursor to first non-white space in line marked by *x*

Line positioning

H	top line on screen
L	last line on screen
M	middle line on screen
+	next line, at first non-white
–	previous line, at first non-white
CR	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

Character positioning

^	first non white-space character
0	beginning of line
$	end of line
h or →	forward
l or ←	backward
^H	same as ← (backspace)
space	same as → (space bar)
f*x*	find next *x*
F*x*	find previous x
t*x*	move to character prior to next *x*
T*x*	move to character following previous *x*
;	repeat last f F t or T
,	repeat inverse of last f F t or T
n|	move to column *n*
%	find matching ({) or }

Words, sentences, paragraphs

w	forward a word
b	back a word
e	end of word

)	to next sentence
}	to next paragraph
(	back a sentence
{	back a paragraph
W	forward a blank-delimited word
B	back a blank-delimited word
E	end of a blank-delimited word

Corrections during insert

^H	erase last character (backspace)
^W	erase last word
erase	your erase character, same as ^H (backspace)
kill	your kill character, erase this line of input
\	quotes your erase and kill characters
ESC	ends insertion, back to command mode
DEL	interrupt, terminates insert mode
^D	backtab one character; reset left margin of *autoindent*
^^D	caret (^) followed by control-d (^D); backtab to beginning of line; do not reset left margin of *autoindent*
0^D	backtab to beginning of line; reset left margin of *autoindent*
^V	quote non-printable character

Insert and replace

a	append after cursor
A	append at end of line
i	insert before cursor
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R*text*ESC	replace characters

Operators

Operators are followed by a cursor motion, and affect all text that would have been moved over. For example, since **w** moves over a word, **dw** deletes the word that would be moved over. Double the operator, e.g., **dd** to affect whole lines.

d	delete
c	change

y	yank lines to buffer
<	left shift
>	right shift
!	filter through command

Miscellaneous Operations

C	change rest of line (**c$**)
D	delete rest of line (**d$**)
s	substitute chars (**cl**)
S	substitute lines (**cc**)
J	join lines
x	delete characters (**dl**)
X	delete characters before cursor (**dh**)
Y	yank lines (**yy**)

Yank and Put

Put inserts the text most recently deleted or yanked; however, if a buffer is named (using the ASCII lower-case letters **a - z**), the text in that buffer is put instead:

3yy	yank 3 lines
3yl	yank 3 characters
p	put back text after cursor
P	put back text before cursor
"*x*p	put from buffer *x*
"*x*y	yank to buffer *x*
"*x*d	delete into buffer *x*

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d*p	retrieve *d*'th last delete

AUTHOR

vi and *ex* were developed by the University of California, Berkeley, California, Computer Science Division, Department of Electrical Engineering and Computer Science.

FILES

/tmp	default directory where temporary work files are placed; it can be changed using the directory option (see the *ex(1) set* command)

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

/usr/lib/.COREterm/?/* subset of compiled terminal description database

NOTES

Two options, although they continue to be supported, have been replaced in the documentation by options that follow the Command Syntax Standard (see *intro*(1)). A **–r** option that is not followed with an option-argument has been replaced by **–L** and **+***command* has been replaced by **–c** *command*.

SEE ALSO

ed(1), edit(1), ex(1).
User's Guide.
Editing Guide.
Curses/Terminfo chapter of the *Programmer's Guide*.

WARNINGS

The encryption options are provided as a separate package only to source customers in the United States.

Tampering with entries in **/usr/lib/.COREterm/?/*** or **/usr/lib/terminfo/?/*** (e.g., changing or removing an entry) can affect programs such as *vi*(1) that expect the entry to be present and correct. In particular, removing the "dumb" terminal may cause unexpected problems.

BUGS

Software tabs using ^T work only immediately after the *autoindent*.

Left and right shifts on intelligent terminals do not make use of insert and delete character operations in the terminal.

NAME

wait – await completion of process

SYNOPSIS

wait [*n*]

DESCRIPTION

Wait for your background process whose process id is *n* and report its termination status. If *n* is omitted, all your shell's currently active background processes are waited for and the return code will be zero.

The shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(1).

CAVEAT

If you get the error message **`cannot fork, too many processes`**, try using the *wait*(1) command to clean up your background processes. If this does not help, the system process table is probably full or you have too many active foreground processes. (There is a limit to the number of process ids associated with your login, and to the number the system can keep track of.)

BUGS

Not all the processes of a 3- or more-stage pipeline are children of the shell, thus cannot be waited for.

If *n* is not an active process id, all your shell's currently active background processes are waited for and the return code will be zero.

NAME

wall – write to all users

SYNOPSIS

/etc/wall

DESCRIPTION

wall reads its standard input until an EOF. It then sends this message to all currently logged-in users preceded by:

```
Broadcast Message from ...
```

It is used to warn all users, typically before shutting down the system.

The sender must be superuser to override any protections the users may have invoked (see *mesg*(1)).

FILES

/dev/tty*

SEE ALSO

mesg(1), write(1).

DIAGNOSTICS

"Cannot send to ..." when the open on a user's TTY file fails.

NAME

wc – word count

SYNOPSIS

wc [**–lwc**] [*names*]

DESCRIPTION

wc counts lines, words, and characters in the named files, or in the standard input if no *names* appear. It also keeps a total count for all named files. A word is a maximal string of characters delimited by spaces, tabs, or newlines.

The options **l**, **w**, and **c** may be used in any combination to specify that a subset of lines, words, and characters are to be reported. The default is **–lwc**.

When *names* are specified on the command line, they will be printed along with the counts.

NAME

what – identify SCCS files

SYNOPSIS

what [**–s**] *files*

DESCRIPTION

what searches the given files for all occurrences of the pattern that *get*(1) substitutes for %Z% (this is **@(#)** at this printing) and prints out what follows until the first ', >, newline, \, or **NULL** character. For example, if the C program in file **f.c** contains

```
char ident[] = "@(#)identification information";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command:

what f.c f.o a.out

will print:

f.c:
identification information

f.o:
identification information

a.out:
identification information

what is intended to be used with the command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually. Only one option exists:

–s Quit after finding the first occurrence of pattern in each file.

SEE ALSO

get(1).
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Exit status is 0 if any matches are found, otherwise 1. Use *help*(1) for explanations.

BUGS

It is possible that an unintended occurrence of the pattern **@(#)** could be found just by chance, but this causes no harm in nearly all cases.

NAME

who – who is on the system

SYNOPSIS

who [**–uTlHqpdbrtas**] [**–n** *x*] [*file*]

who am i

who am I

DESCRIPTION

who can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process-ID of the command interpreter (shell) for each current system user. It examines the **/etc/utmp** file at login time to obtain its information. If *file* is given, that file (which must be in *utmp*(4) format) is examined. Usually, *file* will be **/etc/wtmp**, which contains a history of all the logins since the file was last created.

who with the **am i** or **am I** option identifies the invoking user.

The general format for output is:

name [state] line time [idle] [pid] [comment] [exit]

The *name*, *line*, and *time* information is produced by all options except **–q**; the *state* information is produced only by **–T**; the *idle* and *pid* information is produced only by **–u** and **–l**; and the *comment* and *exit* information is produced only by **–a**. The information produced for **–p**, **–d**, and **–r** is explained during the discussion of each option, below.

With options, *who* can list logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the *init* process. These options are:

–u

lists only those users who are currently logged in. The *name* is the user's login name. The *line* is the name of the line as found in the directory **/dev**. The *time* is the time that the user logged in. The *idle* column contains the number of hours and minutes since activity last occurred on that particular line. A dot (**.**) indicates that the terminal has seen activity in the last minute and is therefore "current". If more than 24 hours have elapsed or the line has not been used since boot time, the entry is marked **old**. This field is useful when trying to determine if a person is working at the terminal. The *pid* is the process-ID of the user's shell. The *comment* is the comment field associated with this line as found in **/etc/inittab** (see *inittab*(4)). This can

contain information about where the terminal is located, the telephone number of the dataset, type of terminal if hard-wired.

–T
is the same as the **–s** option, except that the *state* of the terminal line is printed. The *state* describes whether someone else can write to that terminal. A + appears if the terminal is writable by anyone; a – appears if it is not. **root** can write to all lines having a + or a – in the *state* field. If a bad line is encountered, a **?** is printed.

–l
lists only those lines on which the system is waiting for someone to login. The *name* field is **LOGIN** in such cases. Other fields are the same as for user entries except that the *state* field does not exist.

–H
prints column headings above the regular output.

–q
This is a quick *who*, displaying only the names and the number of users currently logged on. When this option is used, all other options are ignored.

–p
lists any other process which is currently active and has been previously spawned by *init*. The *name* field is the name of the program executed by *init* as found in **/etc/inittab**. The *state*, *line*, and *idle* fields have no meaning. The *comment* field shows the *id* field of the line from **/etc/inittab** that spawned this process. See *inittab*(4).

–d
displays all processes that have expired and not been respawned by *init*. The *exit* field appears for dead processes and contains the termination and exit values (as returned by *wait*(2)), of the dead process. This can be useful in determining why a process terminated.

–b
indicates the time and date of the last reboot.

–r
indicates the current *run-level* of the *init* process. In addition, it produces the process termination status, process id, and process exit status (see *utmp*(4)) under the *idle*, *pid*, and *comment* headings, respectively.

–t
indicates the last change to the system clock (via the *date*(1) command) by **root**. See *su*(1).

–a
processes **/etc/utmp** or the named *file* with all options turned on.

–s
is the default and lists only the *name*, *line*, and *time* fields.

–n *x*
takes a numeric argument, *x*, which specifies the number of users to display per line. *x* must be at least **1**. The –n option must be used with **–q**.

Note to the superuser: after a shutdown to the single-user state, *who* returns a prompt. The reason the prompt displays is because **/etc/utmp** is updated at login time and there is no login in single-user state, therefore, *who* cannot report accurately on this state. *who am i*, however, returns the correct information.

FILES

/etc/utmp
/etc/wtmp
/etc/inittab

SEE ALSO

date(1), login(1), mesg(1), su(1M).
init(1M), inittab(4), utmp(4) in the *System Administrator's Reference Manual.*
wait(2) in the *Programmer's Reference Manual.*

NAME

write – write to another user

SYNOPSIS

write *user* [*line*]

DESCRIPTION

write copies lines from your terminal to that of another user. When first called, it sends the message:

Message from *yourname* **(tty??)** [*date*]...

to the person you want to talk to. When it has successfully completed the connection, it also sends two bells to your own terminal to indicate that what you are typing is being sent.

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal, an interrupt is sent, or the recipient has executed "mesg n". At that point *write* writes EOT on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *line* argument may be used to indicate which line or terminal to send to (e.g., **tty00**); otherwise, the first writable instance of the user found in **/etc/utmp** is assumed and the following message posted:

user is logged on more than one place.
You are connected to "*terminal*".
Other locations are:
terminal

Permission to write may be denied or granted by use of the *mesg*(1) command. Writing to others is normally allowed by default. Certain commands, e.g., *pr*(1), disallow messages to prevent interference with their output. However, if the user has superuser permissions, messages can be forced onto a write-inhibited terminal.

If the character **!** is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first *write* to another user, wait for them to *write* back before starting to send. Each person should end a message with a distinctive signal (i.e., **(o)** for "over") so that the other person knows when to reply. The signal **(oo)** (for "over and out") is suggested when conversation is to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute !

SEE ALSO

mail(1), mesg(1), pr(1), sh(1), who(1)

DIAGNOSTICS

"user is not logged on"
if the person you are trying to *write* to is not logged on.

"Permission denied"
if the person you are trying to *write* to denies that permission (with *mesg*).

"Warning: cannot respond, set mesg -y"
if your terminal is set to *mesg n* and the recipient cannot respond to you.

"Can no longer write to user"
if the recipient has denied permission (*mesg n*) after you had started writing.

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [*flags*] [*command* [*initial-arguments*]]

DESCRIPTION

xargs combines the fixed *initial-arguments* with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the flags specified.

command, which may be a shell file, is searched for, using your $PATH. If *command* is omitted, **/bin/echo** is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or newlines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings, a backslash (\) will escape the next character.

Each argument list is constructed starting with the *initial-arguments*, followed by some number of arguments read from standard input (Exception: see **–i** flag). Flags **–i**, **–l**, and **–n** determine how arguments are selected for each command invocation. When none of these flags are coded, the *initial-arguments* are followed by arguments read continuously from standard input until an internal buffer is full, then *command* is executed with the accumulated args. This process is repeated until there are no more args. When there are flag conflicts (e.g., **–l** vs. **–n**), the last flag has precedence. *flag* values are:

–l*number*

command is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first newline *unless* the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If *number* is omitted, 1 is assumed. Option **–x** is forced.

–i*replstr*

Insert mode: *command* is executed for each line from standard input, taking the entire line as a single arg, inserting it in *initial-arguments*

for each occurrence of *replstr*. A maximum of 5 arguments in *initial-arguments* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters, and option **–x** is also forced. **{}** is assumed for *replstr* if not specified.

–n*number*
Execute *command* using as many standard input arguments as possible, up to *number* arguments maximum. Fewer arguments will be used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* arguments remaining. If option **–x** is also coded, each *number* arguments must fit in the *size* limitation, else *xargs* terminates execution.

–t
Trace mode: The *command* and each constructed argument list are echoed to file descriptor 2 just prior to their execution.

–p
Prompt mode: The user is asked whether to execute *command* each invocation. Trace mode (**–t**) is turned on to print the command instance to be executed, followed by a **?...** prompt. A reply of y (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of *command*.

–x
Causes *xargs* to terminate if any argument list would be greater than *size* characters; **–x** is forced by the options **–i** and **–l**. When neither of the options **–i**, **–l**, or **–n** are coded, the total length of all arguments must be within the *size* limit.

–s*size*
The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If **–s** is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

–eeofstr

eofstr is taken as the logical **EOF** string. Underbar (_) is assumed for the logical **EOF** string if **–e** is not coded. The value **–e** with no *eofstr* coded turns off the logical **EOF** string capability (underbar is taken literally). *xargs* reads standard input until either EOF or the logical **EOF** string is encountered.

xargs will terminate if either it receives a return code of **–1** from, or if it cannot execute, *command*. When *command* is a shell program, it should explicitly *exit* (see *sh*(1)) with an appropriate value to avoid accidentally returning with **–1**.

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

ls $1 | xargs –i –t mv $1/{} $2/{}

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the end of file *log*:

(logname; date; echo $0 $*) | xargs >>log

The user is asked which files in the current directory are to be archived and archives them into *arch* (1.) one at a time, or (2.) many at a time.

1. **ls | xargs –p –l ar r arch**
2. **ls | xargs –p –l | xargs ar r arch**

The following will execute *diff*(1) with successive pairs of arguments originally typed as shell arguments:

echo $* | xargs –n2 diff

SEE ALSO

sh(1).

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–vdlt**] *grammar*

DESCRIPTION

The *yacc* command converts a context-free grammar into a set of tables for a simple automaton which executes an **LR**(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, **y.tab.c**, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *lex*(1) is useful for creating lexical analyzers usable by *yacc*.

If the **–v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

If the **–l** flag is given, the code produced in **y.tab.c** will *not* contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

Runtime debugging code is always generated in **y.tab.c** under conditional compilation control. By default, this code is not included when **y.tab.c** is compiled. However, when *yacc*'s **–t** option is used, this debugging code will be compiled by default. Independent of whether the **–t** option was used, the runtime debugging code is under the control of **YYDEBUG**, a preprocessor symbol. If **YYDEBUG** has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	
yacc.debug, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).
Programmer's Guide.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

CAVEAT

Because file names are fixed, at most one *yacc* process can be active in a given directory at a given time.

NAME

ct – spawn getty to a remote terminal

SYNOPSIS

ct [**–w***n*] [**–x***n*] [**–h**] [**–v**] [**–s***speed*] *telno* ...

DESCRIPTION

ct dials the telephone number of a modem that is attached to a terminal, and spawns a *getty* process to that terminal. The *telno* argument is a telephone number, with equal signs for secondary dial tones and minus signs for delays at appropriate places. (The set of legal characters for *telno* is 0 thru 9, -, =, *, and #. The maximum length *telno* is 31 characters). If more than one telephone number is specified, *ct* will try each in succession until one answers; this is useful for specifying alternate dialing paths.

ct will try each line listed in the file **/usr/lib/uucp/Devices** until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, *ct* will ask if it should wait for one, and if so, for how many minutes it should wait before it gives up. *ct* will continue to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the **–w***n* option, where *n* is the maximum number of minutes that *ct* is to wait for a line.

The **–x***n* option is used for debugging; it produces a detailed output of the program execution on **stderr**. The debugging level, *n*, is a single digit; **–x9** is the most useful value.

Normally, *ct* will hang up the current line so the line can answer the incoming call. The **–h** option prevents this action. The **–h** option also waits for the termination of the specified *ct* process before returning control to the user's terminal. If the **–v** option is used, *ct* sends a running narrative to the standard error output stream.

The data rate may be set with the **–s** option, where *speed* is expressed in baud. The default rate is 1200.

After the user on the destination terminal logs out, there are two things that could occur depending on what type of *getty* is on the line (*getty* or *uugetty*). For the first case, *ct* prompts, Reconnect? If the response begins with the letter **n**, the line will be dropped; otherwise, *getty* will be started again and the `login:` prompt will be printed. In the second case, there is already a *getty* (*uugetty*) on the line, so the `login:` message will appear.

To log out properly, the user must type **CTRL-d**.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

FILES

/usr/lib/uucp/Devices
/usr/adm/ctlog
/usr/spool/locks/LCK*

SEE ALSO

cu(1C), login(1), uucp(1C).
getty(1M), uugetty(1M) in the *System Administrator's Reference Manual.*

BUGS

For a shared port, one used for both dial-in and dial-out, the *uugetty* program running on the line must have the **–r** option specified (see *uugetty*(1M)).

NAME

cu – call another UNIX system

SYNOPSIS

cu [**–s***speed*] [**–l**line] [**–h**] [**–t**] [**–d**] [**–o** | **–e**] [**–n**] *telno*
cu [**–s** *speed*] [**–h**] [**–d**] [**–o** | **–e**] **–l** *line*
cu [**–h**] [**–d**] [**–o** | **–e**] *systemname*

DESCRIPTION

cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files.

cu accepts the following options and arguments:

–s*speed*
specifies the transmission speed (300, 1200, 2400, 4800, 9600). The default value is "Any" speed which will depend on the order of the lines in the **/usr/lib/uucp/Devices** file.

–l*line*
specifies a device name to use as the communication line. This can be used to override the search that would otherwise take place for the first available line having the right speed.

When the **–l** option is used without the **–s** option, the speed of a line is taken from the **Devices** file. When the **–l** and **–s** options are both used together, *cu* will search the **Devices** file to check if the requested speed for the requested line is available. If so, the connection is made at the requested speed; otherwise, an error message is printed and the call will not be made. The specified device is generally a directly connected asynchronous line (e.g., **/dev/tty***ab*) in which case a telephone number (*telno*) is not required. The specified device need not be in the **/dev** directory. If the specified device is associated with an auto dialer, a telephone number must be provided. Use of this option with *systemname* rather than *telno* will not give the desired result (see *systemname* below).

–h
emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

–t
Used to dial an ASCII terminal which has been set to auto answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.

–d
Causes diagnostic traces to be printed.

–o
Designates that odd parity is generated for data sent to the remote system.

–n
For added security, will prompt the user to provide the telephone number to be dialed rather than taking it from the command line.

–e
Designates that even parity is generated for data sent to the remote system.

telno
When using an automatic dialer, the argument is the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of 4 seconds.

systemname
A uucp system name may be used rather than a telephone number; in this case, *cu* will obtain an appropriate direct line or telephone number from **/usr/lib/uucp/Systems.** Note that the *systemname* option should not be used with the **–l** and **–s** options because *cu* will connect to the first available line for the system name specified, ignoring the requested line and speed.

After making the connection, *cu* runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following user initiated commands:

~.
terminate the conversation.

~!
escape to an interactive shell on the local system.

~!*cmd* . . .
run *cmd* on the local system (via **sh –c**).

~$*cmd* . . .
run *cmd* locally and send its output to the remote system.

~%cd
change the directory on the local system. Note that **~!cd** will cause the command to be run by a sub-shell, probably not what was intended.

~%take *src* [*dest*]
copy file *src* (on the remote system) to file *dest* on the local system. If *dest* is omitted, the *src* argument is used in both places.

~%put *src* [*dest*]
copy file *src* (on local system) to file *dest* on remote system. If *dest* is omitted, the *src* argument is used in both places.

For both **~%take** and **put** commands, as each block of the file is transferred, consecutive single digits are printed to the terminal.

~~ *line*
send the line **~** *line* to the remote system.

~%break
transmit a BREAK to the remote system (which can also be specified as **~%b**).

~%debug
toggles the **-d** debugging option on or off (which can also be specified as **~%d**).

~t
prints the values of the termio structure variables for the user's terminal (useful for debugging).

~l
prints the values of the termio structure variables for the remote communication line (useful for debugging).

C)

~%nostop
toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one that does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output. Internally the program accomplishes this by initiating an output diversion to a file when a line from the remote begins with ~.

Data from the remote is diverted (or appended, if >> is used) to *file* on the local system. The trailing ~> marks the end of the diversion.

The use of **~%put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current control characters on the local system. Backslashes are inserted at appropriate places.

The use of **~%take** requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, *tabs* mode (see *stty*(1)) should be set on the remote system if tabs are to be copied without being expanded to spaces.

When *cu* is used on system X to connect to system Y and subsequently used on system Y to connect to system Z, commands on system Y can be executed by using ~~. Executing a *tilde* command reminds the user of the local system uname. For example, uname can be executed on Z, X, and Y as follows:

```
uname
Z
~[X]!uname
X
~~[Y]!uname
Y
```

In general, ~ causes the command to be executed on the original machine, ~~ causes the command to be executed on the next machine in the chain.

EXAMPLES

To dial a system whose telephone number is 9 201 555 1212 using 1200 baud (where dialtone is expected after the 9):

```
cu -s1200 9=12015551212
```

If the speed is not specified, "Any" is the default value.

To login to a system connected by a direct line:

cu –l /dev/ttyXX

or

cu –l ttyXX

To dial a system with the specific line and a specific speed:

cu –s1200 –l ttyXX

To dial a system using a specific line associated with an auto dialer:

cu –l culXX 9=12015551212

To use a system name:

cu systemname

FILES

/usr/lib/uucp/Systems
/usr/lib/uucp/Devices
/usr/spool/locks/LCK..(tty-device)

SEE ALSO

cat(1), ct(1C), echo(1), stty(1), uucp(1C), uname(1).

DIAGNOSTICS

Exit code is zero for normal exit, otherwise, one.

WARNINGS

The *cu* command does not do any integrity checking on data it transfers. Data fields with special *cu* characters may not be transmitted properly. Depending on the interconnection hardware, it may be necessary to use a **~.** to terminate the conversion even if **stty 0** has been used. Non-printing characters are not dependably transmitted using either the **~%put** or **~%take** commands. *cu* between an IMBR1 and a penril modem will not return a login prompt immediately upon connection. A carriage return will return the prompt.

BUGS

There is an artificial slowing of transmission by *cu* during the ~%put operation so that loss of data is unlikely.

NAME

uucp, uulog, uuname – UNIX-to-UNIX system copy

SYNOPSIS

uucp [*options*] *source-files destination-file*
uulog [*options*] **–s***system*
uulog [*options*] *system*
uulog [*options*] **–f***system*
uuname [**–l**] [**–c**]

DESCRIPTION

uucp

uucp copies files named by the *source-file* arguments to the *destination-file* argument. A file name may be a pathname on your machine, or may have the form:

system-name!path-name

where *system-name* is taken from a list of system names that *uucp* knows about. The *system-name* may also be a list of names:

system-name!system-name!...!system-name!path-name

in which case an attempt is made to send the file via the specified route, to the destination. See **WARNINGS** and **BUGS** for restrictions. Care should be taken to ensure that intermediate nodes in the route are willing to forward information (see **WARNINGS** for restrictions).

The following shell metacharacters are disallowed in *system-name*:

' ; & | ^ < > ( ) <CR> <TAB> <SPACE>

Pathnames may be one of the following:

(1) a full pathname.

(2) a pathname preceded by *~user* where *user* is a login name on the specified system and is replaced by that user's login directory.

(3) a pathname preceded by *~/destination* where *destination* is appended to **/usr/spool/uucppublic**. Note that this destination will be treated as a file name unless more than one file is being transferred by this request or the destination is already a directory. To ensure that it is a directory, follow the destination with a '/'. For example, ~/dan/ as the destination will make the directory **/usr/spool/uucppublic/dan** if it does not exist and put the requested file(s) in that directory.

(4) anything else is prefixed by the current directory.

If the result is an erroneous pathname for the remote system the copy will fail. If the *destination-file* is a directory, the last part of the *source-file* name is used.

uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*:

–c
Do not copy local file to the spool directory for transfer to the remote machine (default).

–C
Force the copy of local files to the spool directory for transfer.

–d
Make all necessary directories for the file copy (default).

–f
Do not make intermediate directories for the file copy.

–g*grade*
grade is a single letter/number; lower ASCII sequence characters will cause the job to be transmitted earlier during a particular conversation.

–j
Output the job identification ASCII string on the standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.

–m
Send mail to the requester when the copy is completed.

–n*user*
Notify *user* on the remote system that a file was sent.

–r
Do not start the file transfer, just queue the job.

–s*file*
Report status of the transfer to *file*. Note that the *file* must be a full pathname.

–x*debug_level*
Produce debugging output on standard output. The *debug_level* is a number between 0 and 9; higher numbers give more detailed information. (Debugging will not be available if **uucp** was compiled with –DSMALL.)

uulog
uulog queries a log file of *uucp* or *uuxqt* transactions in a file **/usr/spool/uucp/.Log/uucico/***system* or **/usr/spool/uucp/.Log/uuxqt/***system*.

The options cause *uulog* to print logging information:

–s*sys*
Print information about file transfer work involving system *sys*.

–f*system*
Does a "tail –f" of the file transfer log for *system*. (You must press **BREAK** to exit this function.) The following are other options used with the above:

–**x**
Look in the *uuxqt* log file for the given system.

–*number*
Indicates that a "tail" command of *number* lines should be executed.

uuname
uuname lists the names of systems known to *uucp*. The –**c** option returns the names of systems known to *cu*. (The two lists are the same, unless your machine is using different **Systems** files for *cu* and *uucp*. See the **Sysfiles** file.) The –**l** option returns the local system name.

FILES

/usr/spool/uucp	spool directories
/usr/spool/uucppublic/*	public directory for receiving and sending (**/usr/spool/uucppublic**)
/usr/lib/uucp/*	other data and program files

SEE ALSO
mail(1), uustat(1C), uux(1C).
uuxqt(1M) in the *System Administrator's Reference Manual*.
chmod(2) in the *Programmer's Reference Manual*.

1C)

WARNINGS

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will probably not be able to fetch files by pathname; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary pathnames. As distributed, the remotely accessible files are those whose names begin **/usr/spool/uucppublic** (equivalent to ~/).

All files received by *uucp* will be owned by *uucp*.

The **–m** option will only work sending files or receiving a single file. Receiving multiple files specified by special shell characters **? * [...]** will not activate the **–m** option.

The forwarding of files through other systems may not be compatible with the previous version of *uucp*. If forwarding is used, all systems in the route must have the same version of *uucp*.

BUGS

Protected files and files that are in protected directories that are owned by the requester can be sent by *uucp* using the **–C** option. However, if the requestor is **root**, and the directory is not searchable by "other" or the file is not readable by "other", the request will fail.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

uustat [–a]
uustat [–m]
uustat [–p]
uustat [–q]
uustat [–k*jobid*]
uustat [–r*jobid*]
uustat [–s*system*] [**–u***user*]

DESCRIPTION

uustat will display the status of, or cancel, previously specified *uucp* commands, or provide general status on *uucp* connections to other systems. Only one of the following options can be specified with *uustat* per command execution:

–a
Output all jobs in queue.

–m
Report the status of accessibility of all machines.

–p
Execute a "ps –flp" for all the process-ids that are in the lock files.

–q
List the jobs queued for each machine. If a status file exists for the machine, its date, time and status information are reported. In addition, if a number appears in () next to the number of **C** or **X** files, it is the age in days of the oldest **C./X.** file for that system. The Retry field represents the number of hours until the next possible call. The Count is the number of failure attempts. Note that for systems with a moderate number of outstanding jobs, this could take 30 seconds or more of real-time to execute. As an example of the output produced by the **–q** option:

```
eagle      3C    04/07-11:07NO DEVICES AVAILABLE
mh3bs3     2C    07/07-10:42SUCCESSFUL
```

This output tells how many command files are waiting for each system. Each command file may have zero or more files to be sent (zero means to call the system and see if work is to be done). The date and time refer to the previous interaction with the system followed by the status of the interaction.

–k*jobid*
Kill the *uucp* request whose job identification is *jobid*. The killed *uucp* request must belong to the person issuing the *uustat* command unless one is the super-user.

–r*jobid*
Rejuvenate *jobid*. The files associated with *jobid* are touched so that their modification time is set to the current time. This prevents the cleanup daemon from deleting the job until the jobs modification time reaches the limit imposed by the deamon.

Either or both of the following options can be specified with *uustat*:

–s*sys*
Report the status of all *uucp* requests for remote system *sys*.

–u*user*
Report the status of all *uucp* requests issued by *user*.

Output for both the **–s** and **–u** options has the following format:

```
eaglen0000      4/07-11:01:03 (POLL)
eagleN1bd7      4/07-11:07    Seagledan522 /usr/dan/A
eagleC1bd8      4/07-11:07    Seagledan59 D.3b2al2ce4924
                4/07-11:07    Seagledanrmail mike
```

With the above two options, the first field is the *jobid* of the job. This is followed by the date/time. The next field is either an 'S' or 'R' depending on whether the job is to send or request a file. This is followed by the user-id of the user who queued the job. The next field contains the size of the file, or in the case of a remote execution (*rmail* - the command used for remote mail), the name of the command. When the size appears in this field, the file name is also given. This can either be the name given by the user or an internal name (e.g., D.3b2alce4924) that is created for data files associated with remote executions (*rmail* in this example).

When no options are given, *uustat* outputs the status of all *uucp* requests issued by the current user.

FILES

/usr/spool/uucp/* spool directories

SEE ALSO

uucp(1C).

NAME

uuto, uupick – public UNIX-to-UNIX system file copy

SYNOPSIS

uuto [*options*] *source-files destination*
uupick [**–s** *system*]

DESCRIPTION

uuto sends *source-files* to *destination*. *uuto* uses the *uucp*(1C) facility to send files, while it allows the local system to control the file access. A source-file name is a pathname on your machine. Destination has the form:

system!*user*

where *system* is taken from a list of system names that *uucp* knows about (see *uuname*). *user* is the login name of someone on the specified system.

Two *options* are available:

–p
Copy the source file into the spool directory before transmission.

–m
Send mail to the sender when the copy is complete.

The files (or sub-trees if directories are specified) are sent to **PUBDIR** on *system*, where **PUBDIR** is a public directory defined in the *uucp* source. By default, this directory is **/usr/spool/uucppublic**. Specifically, the files are sent to:

PUBDIR/receive/*user***/***mysystem***/files.**

The destined recipient is notified by *mail*(1) of the arrival of files.

uupick accepts or rejects the files transmitted to the user. Specifically, *uupick* searches **PUBDIR** for files destined for the user. For each entry (file or directory) found, the following message is printed on the standard output:

from *system*: [file *file-name*] [dir *dirname*] **?**

uupick then reads a line from the standard input to determine the disposition of the file:

<newline>
Go on to next entry.

d
Delete the entry.

C)

m [*dir*]
Move the entry to named directory *dir*. If *dir* is not specified as a complete pathname (in which $HOME is legitimate), a destination relative to the current directory is assumed. If no destination is given, the default is the current directory.

a [*dir*]
Same as **m** except moving all the files sent from *system*.

p
Print the content of the file.

q
Stop.

EOT (**CTRL**)
Same as **q**.

!*command*
Escape to the shell to do *command*.

*
Print a command summary.

uupick invoked with the **-s***system* option will only search the **PUBDIR** for files sent from *system*.

FILES

PUBDIR/usr/spool/uucppublic public directory

SEE ALSO

mail(1), uucp(1C), uustat(1C), uux(1C).
uucleanup(1M) in the *System Administrator's Reference Manual*.

WARNINGS

To send files that begin with a dot (e.g., **.profile**), the files must by qualified with a dot. For example: **.profile, .prof***, **.profil?** are correct; whereas ***prof***, **?profile** are incorrect.

NAME

uux – UNIX-to-UNIX system command execution

SYNOPSIS

uux [*options*] *command-string*

DESCRIPTION

uux will gather zero or more files from various systems, execute a command on a specified system and then send standard output to a file on a specified system.

NOTE: For security reasons, most installations limit the list of commands executable on behalf of an incoming request from *uux*, permitting only the receipt of mail (see *mail*(1)). (Remote execution permissions are defined in **/usr/lib/uucp/Permissions**.)

The *command-string* is made up of one or more arguments that look like a shell command line, except that the command and file names may be prefixed by *system-name*!. A `NULL` *system-name* is interpreted as the local system.

File names may be one of:

(1) a full pathname;

(2) a pathname preceded by ~*xxx* where *xxx* is a login name on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

As an example, the command:

uux "!diff usg!/usr/dan/file1 pwba!/a4/dan/file2 !~/dan/file.diff"

will get the *file1* and *file2* files from the "usg" and "pwba" machines, execute a *diff*(1) command and put the results in *file.diff* in the local **PUBDIR/dan/directory**.

Any special shell characters such as **<>;|** should be quoted either by quoting the entire *command-string*, or quoting the special characters as individual arguments.

uux will attempt to get all files to the execution system. For files that are output files, the file name must be escaped using parentheses. For example, the command:

uux a!cut -f1 b!/usr/file \(c!/usr/file\)

gets **/usr/file** from system "b" and sends it to system "a", performs a *cut* command on that file and sends the result of the *cut* command to system "c".

uux will notify you if the requested command on the remote system was disallowed. This notification can be turned off by the **–n** option. The response comes by remote mail from the remote machine.

The following *options* are interpreted by *uux*:

–
The standard input to *uux* is made the standard input to the *command-string*.

–a*name*
Use *name* as the user identification replacing the initiator user-id. (Notification will be returned to the user.)

–b
Return whatever standard input was provided to the *uux* command if the exit status is non-zero.

–c
Do not copy local file to the spool directory for transfer to the remote machine (default).

–C
Force the copy of local files to the spool directory for transfer.

–g*grade*
grade is a single letter/number; lower ASCII sequence characters will cause the job to be transmitted earlier during a particular conversation.

–j
Output the jobid ASCII string on the standard output which is the job identification. This job identification can be used by *uustat* to obtain the status or terminate a job.

–n
Do not notify the user if the command fails.

–p
Same as –: The standard input to *uux* is made the standard input to the *command-string*.

–r
Do not start the file transfer, just queue the job.

–s*file*
Report status of the transfer in *file.*

–x*debug_level*
Produce debugging output on the standard output. The *debug_level* is a number between 0 and 9; higher numbers give more detailed information.

–z
Send success notification to the user.

FILES

/usr/lib/uucp/spool	spool directories
/usr/lib/uucp/Permissions	remote execution permissions
/usr/lib/uucp/*	other data and programs

SEE ALSO

cut(1), mail(1), uucp(1C), uustat(1C).

WARNINGS

Only the first command of a shell pipeline may have a *system-name*!. All other commands are executed on the system of the first command.

The use of the shell metacharacter * will probably not do what you want it to do. The shell tokens << and >> are not implemented.

The execution of commands on remote systems takes place in an execution directory known to the *uucp* system. All files required for the execution will be put into this directory unless they already reside on that machine. Therefore, the simple file name (without path or machine reference) must be unique within the *uux* request. The following command will not work:

uux "a!diff b!/usr/dan/xyz c!/usr/dan/xyz > !xyz.diff"

but the command:

uux "a!diff a!/usr/dan/xyz c!/usr/dan/xyz > !xyz.diff"

will work (if *diff* is a permitted command).

BUGS

Protected files and files that are in protected directories that are owned by the requestor can be sent in commands using *uux*. However, if the requestor is **root**, and the directory is not searchable by "other", the request will fail.

PERMUTED INDEX

NAME

join – relational database operator

SYNOPSIS

join [*options*] *file1 file2*

DESCRIPTION

join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is –, the standard input is used.

file1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line (see *sort*(1)).

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

The default input field separators are blank, tab, or newline. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the options below use the argument *n*. This argument should be a **1** or a **2** referring to either *file1* or *file2*, respectively. The following options are recognized:

–a*n*
In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

–e *s*
Replace empty output fields by string *s*.

–j*n m*
Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with **1**.

–o *list*
Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number. The common field is not printed unless specifically requested.

–t*c*

Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant. The character *c* is used as the field separator for both input and output.

EXAMPLE

The following command line will join the password file and the group file, matching on the numeric group ID, and outputting the login name, the group name and the login directory. It is assumed that the files have been sorted in ASCII collating sequence on the group ID fields:

join –j1 4 –j2 3 –o 1.1 2.1 1.6 –t: /etc/passwd /etc/group

SEE ALSO

awk(1), comm(1), sort(1), uniq(1)

BUGS

With default field separation, the collating sequence is that of **sort –b**; with **–t**, the sequence is that of a plain sort.

The conventions of *join*, *sort*, *comm*, *uniq* and *awk*(1) are wildly incongruous.

File names that are numeric may cause conflict when the **-o** option is used right before listing file names.

NAME

kill – terminate a process

SYNOPSIS

kill [– *signo*] *PID* ...

DESCRIPTION

kill sends signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with & is reported by the shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using *ps*(1).

The details of the kill are described in *kill*(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The killed process must belong to the current user unless he is the superuser.

If a signal number preceded by – is given as first argument, that signal is sent instead of terminate (see *signal*(2)). In particular "kill –9 . . ." is a sure kill.

SEE ALSO

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

NAME

ksh, rksh – shell, the standard/restricted command programming language

SYNOPSIS

ksh [**–aefhikmnoprstuvx**] [**–o** *option*] . . . [**–c** *string*] [*arg* . . .]
rksh [**–aefhikmnoprstuvx**] [**–o** *option*] . . . [**–c** *string*] [*arg* . . .]

DESCRIPTION

ksh is a command programming language that executes commands read from a terminal or a file. See **Invocation** for the meaning of arguments to the shell. *ksh* may also take the place of the regular shell, *sh* and the restricted shell, *rsh*.

Definitions

A metacharacter is one of the following characters:

; & ( ) | < > newline 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 aliases, functions, and named parameters. A word is a sequence of characters separated by one or more non-quoted metacharacters.

Commands

A *simple-command* is a sequence of blank separated words which may be preceded by a parameter assignment list (see **Environment**). The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec*(2)). The value of a simple-command is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of one or more commands separated by **|** . The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **&&**, or **||**, and optionally terminated by **;**, **&**, or **|&** . Of these five symbols, **;**, **&**, and **|&** have equal precedence, which is lower than that of **&&** and **||** . The symbols **&&** and **||** also have equal precedence. A semicolon (**;**) causes sequential execution of the preceding pipeline; an ampersand (**&**)

causes asynchronous execution of the preceding pipeline (i.e., the shell does not wait for that pipeline to finish). The symbol **|&** causes asynchronous execution of the preceding command or pipeline with a two-way pipe established to the parent shell.

The standard input and output of the spawned command can be written to and read from by the parent Shell using the **–p** option of the special commands **read** and **print** described later. Only one such command can be active at any given time. The symbol **&&** (**||**) causes the list following it to be executed only if the preceding pipeline returns a zero (non-zero) value. An arbitrary number of newlines may appear in a list, instead of semicolons, to delimit commands.

A *command* is either a simple-command or one of the following. Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

for *identifier* [**in** *word* . . .] **do** *list* **done**
Each time a *for* command is executed, *identifier* is set to the next *word* taken from the **in** *word* list. If **in** *word* . . . is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see **Parameter Substitution**). Execution ends when there are no more words in the list.

select *identifier* [**in** *word* . . .] **do** *list* **done**
A *select* command prints on standard error (file descriptor 2), the set of *words*, each preceded by a number. If **in** *word* . . . is omitted, then the positional parameters are used instead (see **Parameter Substitution**). The **PS3** prompt is printed and a line is read from the standard input. If this line consists of 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 this line is empty the selection list is printed again. Otherwise, the value of the parameter *identifier* is set to **NULL**. The contents of the line read from standard input is saved in the parameter **REPLY**. The *list* is executed for each selection until a **BREAK** or EOF is encountered.

case *word* **in** [*pattern* [**|** *pattern*] . . . **)** *list* **;;**] . . . **esac**
A *case* command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see **File Name Generation**).

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns a zero exit status.

while *list* **do** *list* **done**
until *list* **do** *list* **done**
A *while* command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise, the loop terminates. If no commands in the **do** *list* are executed, then the *while* command returns a zero exit status; *until* may be used in place of *while* to negate the loop termination test.

(*list***)**
Execute *list* in a separate environment. Note, that if two adjacent open parentheses are needed for nesting, a space must be inserted to avoid arithmetic evaluation as described below. A parenthesized list used as a command argument denotes *process substitution* as described below.

{ *list***;}**
list is simply executed. Note that **{** is a *keyword* and requires a blank in order to be recognized.

function *identifier* **{** *list* **;}**
identifier **() {** *list* **;}**
Define a function that is referenced by *identifier*. The body of the function is the *list* of commands between **{** and **}**. (See **Functions**.)

time *pipeline*
The *pipeline* is executed and the elapsed time as well as the user and system time are printed on standard error.

The following keywords are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done {} function lect time

Comments

A word beginning with **#** causes that word and all the following characters up to a newline to be ignored.

Aliasing

The first word of each command is replaced by the text of an **alias** if an **alias** for this word has been defined. The first character of an **alias** name can be any non-special printable character, but the rest of the characters must be the same as for a valid *identifier*. The replacement string can contain any valid Shell script including the metacharacters listed above. The first word of each command of the replaced text will not be tested for additional 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 special built-in commands but cannot be used to redefine the keywords listed above. Aliases can be created, listed, and exported with the *alias* command and can be removed with the *unalias* command. Exported aliases remain in effect for sub-shells but must be reinitialized for separate invocations of the Shell (see **Invocation**).

aliasing is performed when scripts are read, not while they are executed. Therefore, for an alias to take effect, the *alias* command has to be executed before the command that references the alias is read.

Aliases are frequently used as a short hand for full path names. An option to the aliasing facility 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 subsequent reference will redefine the value. Several tracked aliases are compiled into the shell. The **–h** option of the **set** command makes each command name which is a valid alias name into a tracked alias.

The following *exported aliases* are compiled into the shell but can be unset or redefined:

```
false='let 0'
functions='typeset -f'
history='fc -l'
integer='typeset -i'
nohup='nohup '
r='fc -e -'
true=':'
type='whence -v'
hash='alias -t'
```

Tilde Substitution

After alias substitution is performed, each word is checked to see if it begins with an unquoted ~. If it does, then the word up to a / is checked to see if it matches a user name in the **/etc/passwd** file. If a match is found, the ~ and the matched login name is 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 ~ by itself, or in front of a /, is replaced by the value of the HOME parameter. A ~ followed by a + or - is replaced by the value of the parameter **PWD** and **OLDPWD** respectively.

In addition, the value of each *keyword parameter* is checked to see if it begins with a ~ or if a ~ appears after a :. In either of these cases, a *tilde* substitution is attempted.

Command Substitution

The standard output from a command enclosed in parenthesis preceded by a dollar sign (**$( )**) or a pair of grave accents (**` `**) may be used as part or all of a word; trailing newlines are removed. In the second (archaic) form, the string between the quotes is processed for special quoting characters before the command is executed. (See **Quoting**.) The command substitution **$(cat file)** can be replaced by the equivalent but faster **$(<file)**. Command substitution of most special commands that do not perform input/output redirection are carried out without creating a separate process.

Process Substitution

This feature is only available on operating systems that support the **/dev/fd** directory for naming open files (not available on SYSTEM V/88). Each command argument of the form **(***list***)**, **<(***list***)**, or **>(***list***)** will run

process *list* asynchronously connected to some file in **/dev/fd**. The name of this file will become the argument to the command. If the form with > is selected then writing on this file will provide input for *list*. If < is used or omitted, then the file passed as an argument will contain the output of the *list* process. For example,

paste (cut –f1 *file1***) (cut –f3** *file2***) | tee >(***process1***) >(***process2***)**

cuts fields 1 and 3 from the files *file1* and *file2* respectively, *pastes* the results together, and sends it to the processes *process1* and *process2*, as well as putting it onto the standard output. Note that the file, which is passed as an argument to the command, is a *pipe*(2) so programs that expect to *lseek*(2) on the file will not work.

Parameter Substitution

A *parameter* is an *identifier*, one or more digits, or any of the characters *****, **@**, **#**, **?**, **–**, **$**, and **!**. A *named parameter* (a parameter denoted by an identifier) has a *value* and zero or more *attributes*. *named parameters* can be assigned *values* and *attributes* by using the **typeset** special command. The attributes supported by the Shell are described later with the **typeset** special command. Exported parameters pass values and attributes to subshells but only values to the environment.

The shell supports a limited one-dimensional array facility. An element of an array parameter is referenced by a *subscript*. A *subscript* is denoted by a **[**, followed by an *arithmetic expression* (see **Arithmetic Evaluation**) followed by a **]**. The value of all subscripts must be in the range of 0 through 511. Arrays need not be declared. Any reference to a named parameter with a valid subscript is legal and an array will be created if necessary. Referencing an array without a subscript is equivalent to referencing the first element.

The *value* of a *named parameter* may also be assigned by writing:

name=*value* [*name*=*value*] . . .

If the integer attribute, **–i**, is set for *name* the *value* is subject to arithmetic evaluation as described below. Positional parameters, parameters denoted by a number, may be assigned values with the **set** special command. Parameter **$0** is set from argument zero when the shell is invoked. The character **$** is used to introduce substitutable *parameters*:

${*parameter*}

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 named parameter is subscripted. If *parameter* is one or more digits then it is a positional parameter. A positional parameter of more than one digit must be enclosed in braces. If *parameter* is * or @, then all the positional parameters, starting with **$1**, are substituted (separately a field separator character). If an array *identifier* with subscript * or @ is used, then the value for each of the elements is substituted (separated by a field separator character).

${#*parameter*}

If *parameter* is * or @, the number of positional parameters is substituted. Otherwise, the length of the value of the *parameter* is substituted.

${#*identifier*[*]}

The number of elements in the array *parameter* is substituted.

${*parameter*:-*word*}

If *parameter* is set and is non-**NULL**, substitute its value; otherwise, substitute *word*.

${*parameter*:=*word*}

If *parameter* is not set or is **NULL**, set it to *word*; the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${*parameter*:?*word*}

If *parameter* is set and is non-**NULL**, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, a standard message is printed.

${*parameter*:+*word*}

If *parameter* is set and is non-**NULL**, substitute *word*; otherwise, substitute nothing.

${*parameter*#*pattern*}

${*parameter*##*pattern*}

If the Shell *pattern* matches the beginning of the value of *parameter*, the value of this substitution is the value of the *parameter* with the matched portion deleted; otherwise, the value of this *parameter* is substituted. In the first form, the smallest matching pattern is deleted; in the latter form, the largest matching pattern is deleted.

${*parameter***%***pattern***}**
${*parameter***%%***pattern***}**
If the Shell *pattern* matches the end of the value of *parameter*, the value of *parameter* with the matched part deleted; otherwise, substitute the value of *parameter*. In the first form, the smallest matching pattern is deleted; in the latter form, the largest matching pattern is deleted.

In the above syntax, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is `NULL`:

echo ${d:-$(pwd)}

If the colon (**:**) is omitted from the above expressions, then the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

#
The number of positional parameters in decimal.

-
Flags supplied to the shell on invocation or by the **set** command.

?
The decimal value returned by the last executed command.

$
The process number of this shell.

_
The last argument of the previous command. This parameter is not set for commands which are asynchronous. This parameter is also used to hold the name of the matching **MAIL** file when checking for mail. Finally, the value of this parameter is set to the full pathname of each program the shell invokes and is passed in the *environment*.

!
The process number of the last background command invoked.

PPID
The process number of the parent of the shell.

PWD
The present working directory set by the **cd** command.

OLDPWD
The previous working directory set by the **cd** command.

RANDOM
Each time this parameter is referenced, a random integer is generated. The sequence of random numbers can be initialized by assigning a numeric value to **RANDOM**.

REPLY
This parameter is set by the **select** statement and by the *read* special command when no arguments are supplied.

SECONDS
Each time this parameter is referenced, the number of seconds since shell invocation is returned. If this parameter 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 parameters are used by the shell:

CDPATH
The search path for the *cd* command.

COLUMNS
If this variable is set, the value is used to define the width of the edit window for the 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, then the corresponding option (see **Special Command set**) will be turned on.

ENV
If this parameter is set, then parameter substitution is performed on the value to generate the pathname of the script that will be executed when the *shell* is invoked. (See **Invocation**.) This file is typically used for *alias* and function definitions.

FCEDIT
The default editor name for the **fc** command.

IFS
Internal field separators, normally **space**, **tab**, and **newline** that is used to separate command words which result from command or parameter substitution and for separating words with the special command *read*. The first character of the **IFS** parameter is used to separate arguments for the **"$*"** substitution (see **Quoting**).

HISTFILE
If this parameter is set when the shell is invoked, then the value is the pathname of the file that will be used to store the command history. (See **Command Re-entry**.)

HISTSIZE
If this parameter is set when the shell is invoked, then the number of previously entered commands that are accessible by this shell will be greater than or equal to this number. The default is 128.

HOME
The default argument (home directory) for the *cd* command.

LINES
If this variable is set, the value is used 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 parameter is set to the name of a mail file *and* the **MAILPATH** parameter is not set, the shell informs the user of arrival of mail in the specified file.

MAILCHECK
This variable specifies how often (in seconds) the shell will check for changes in the modification time of any of the files specified by the **MAILPATH** or **MAIL** parameters. The default value is 600 seconds. When the time has elapsed, the shell will check before issuing the next prompt.

MAILPATH
A colon (**:**) separated list of file names. If this parameter is set, the shell informs the user of any modifications to the specified files that have occurred within the last **MAILCHECK** seconds. Each file name can be followed by a **?** and a message that will be printed. The message will undergo parameter and command substitution with the parameter, **$_** 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**). The user may not change **PATH** if executing under *rksh* (except in *.profile*).

PS1
The value of this parameter is expanded for parameter substitution to define the primary prompt string which by default is "**$** ". The character **!** in the primary prompt string is replaced by the *command* number (see **Command Re-entry**).

PS2
Secondary prompt string, by default "**>** ".

PS3
Selection prompt string used within a **select** loop, by default "**#?** ".

SHELL
The pathname of the *shell* is kept in the environment. At invocation, if the value of this variable contains an **r** in the basename, then the shell becomes restricted.

TMOUT
If set to a value greater than zero, the shell will terminate if a command is not entered within the prescribed number of seconds after issuing the **PS1** prompt. (Note that the shell can be 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* then the corresponding option (see **Special Command set**) will be turned on.

The shell gives default values to **PATH**, **PS1**, **PS2**, **MAILCHECK**, **TMOUT** and **IFS**, while **HOME**, **SHELL ENV** and **MAIL** are not set at all by the shell (although **HOME** *is* set by *login*(1)). On some systems, **MAIL** and **SHELL** are also set by *login*(1)).

Blank Interpretation

After parameter and command substitution, the results of substitutions are scanned for the field separator characters (those found in **IFS**) and split into distinct arguments where such characters are found. Explicit **NULL** arguments (**" "** or **''**) are retained. Implicit **NULL** arguments (those resulting from *parameters* that have no values) are removed.

File Name Generation

Following substitution, each command *word* is scanned for the characters *****, **?**, and **[** unless the **–f** option has been **set**. If one of these characters appears then the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, then the word is left unchanged. When a *pattern* is used for file name generation, the character . at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly. In other instances of pattern matching, the / and . are not treated specially.

*****
Matches any string, including the **NULL** string.

?
Matches any single character.

[...]
Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening "[" is a "!" then any character not enclosed is matched. A – can be included in the character set by putting it as the first or last character.

Quoting

Each of the *metacharacters* listed above (see **Definitions**) has a special meaning to the shell and causes termination of a word unless quoted. A character may be *quoted* (i.e., made to stand for itself) by preceding it with a \. The pair **\new-line** is ignored. All characters enclosed between a pair of single quote marks (''), are quoted.

A single quote cannot appear within single quotes. Inside double quote marks (" "), parameter and command substitution occurs and \ quotes the characters \ , ` , ", and $. The meaning of $* and $@ is identical when not quoted or when used as a parameter assignment value or as a file name. However, when used as a command argument, "$*" is equivalent to "$1*d*$2*d*...", where *d* is the first character of the **IFS** parameter, whereas "$@" is equivalent to "$1" "$2" ... . If the grave quotes, occur within double quotes then \ also quotes the character.

The special meaning of keywords or aliases can be removed by quoting any character of the keyword. The recognition of function names or special command names listed below cannot be altered by quoting them.

Arithmetic Evaluation

An ability to perform integer arithmetic is provided with the special command **let**. 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, then base 10 is used.

An internal integer representation of a *named parameter* can be specified with the **–i** option of the **typeset** special command. When this attribute is selected, the first assignment to the parameter determines the arithmetic base to be used when parameter substitution occurs.

Since many of the arithmetic operators require quoting, an alternative form of the *let* command is provided. For any command that begins with a **((**, all the characters until a matching **))** are treated as a quoted expression. More precisely, **((...))** is equivalent to **let "..."**.

Prompting

When used interactively, the shell prompts with the value of **PS1** before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt (i.e., the value of **PS2**) is issued.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a command and are not passed on to the invoked command. Command and parameter substitution occurs before *word* or *digit* is used except as noted below. File name generation occurs only if the pattern matches a single file and blank interpretation is not performed.

<word

Use file *word* as standard input (file descriptor 0).

>word

Use file *word* as standard output (file descriptor 1). If the file does not exist, it is created; otherwise, it is truncated to zero length.

>>word

Use file *word* as standard output. If the file exists, output is appended to it (by first seeking to the EOF); otherwise, the file is created.

<<[*–*]*word*

The shell input is read up to a line that is the same as *word*, or to an EOF. No parameter substitution, command substitution or file name generation is performed on *word*. The resulting document, called a *here-document*, becomes the standard input.

If any character of *word* is quoted, no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, **\new-line** is ignored, and **** must be used to quote the characters ****, **$**, **`**, and the first character of *word*. If **–** is appended to **<<**, all leading tabs are stripped from *word* and from the document.

<&digit

The standard input is duplicated from file descriptor *digit* (see *dup*(2)). Similarly for the standard output using *>& digit*.

<&–

The standard input is closed. Similarly for the standard output using

>&-.

If one of the above is preceded by a digit, the file descriptor number referred to is that specified by the digit (instead of the default 0 or 1). For example:

... 2>&1

means file descriptor 2 is to be opened for writing as a duplicate of file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*file descriptor*, *file*) association at the time of evaluation. For example:

... 1>*fname* 2>&1

first associates file descriptor 1 with file *fname*. It then associates file descriptor 2 with the file associated with file descriptor 1 (i.e., *fname*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been), then file descriptor 1 would be associated with file *fname*.

If a command is followed by **&** and job control is not active, then the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Environment

The *environment* (see *environ*(7)) is 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 *identifiers* and the values are character strings. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value and marking it *export*. Executed commands inherit the environment. If the user modifies the values of these parameters 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 which must be noted in *export* or *typeset –x* commands.

The environment for any *simple-command* or function may be augmented by prefixing it with one or more parameter assignments. A parameter assignment argument is a word of the form *identifier=value*. Thus:

```
TERM=450 cmd args
```

and

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

are equivalent (as far as the above execution of *cmd* is concerned).

If the **–k** flag is set, *all* parameter assignment arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and then **c:**

```
echo a=b c
set –k
echo a=b c
```

Functions

The **function** keyword, described in the **Commands** section, is used to define 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 the arguments passed as positional parameters. (See **Execution**.)

Functions execute in the same process as the caller and share all files, traps (other than **EXIT** and **ERR**) and present working directory with the caller. A trap set on **EXIT** inside a function is executed after the function completes. Ordinarily, variables are shared between the calling program and the function. However, the *typeset* special command used within a function defines local variables whose scope includes the current function and all functions it calls.

The special command **return** is used to return from function calls. Errors within functions return control to the caller.

Function identifiers can be listed with the **–f** option of the *typeset* special command. The text of functions will also be listed. Function can be undefined with the **–f** option of the *unset* special command.

Ordinarily, functions are unset when the shell executes a shell script. The **-xf** option of the *typeset* command allows a function to be exported to scripts that are executed without a separate invocation of the shell. Functions that need to be defined across separate invocations of the shell should be placed in the **ENV** file.

Jobs

If the **monitor** option of the *set* command is turned on, an interactive shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with **&**, the shell prints a line which looks like:

[1] 1234

indicating that the job that was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

This paragraph and the next require features that are not in all versions of UNIX and may not apply (does not apply to SYSTEM V/88). If you are running a job and wish to do something else you may hit the key ^Z (CTRL-Z), which sends a STOP signal to the current job. The shell will then normally indicate that the job has been 'Stopped,' and print another prompt. You can then manipulate the state of this job, putting it in the background with the **bg** command, or run some other commands and then eventually bring the job back into the foreground with the foreground command **fg**. A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "stty tostop". If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character % introduces a job name. If you wish to refer to job number 1, you can name it as **%1**. Jobs can also be named by prefixes of the string typed in to **kill** or restart them. Thus, on systems that support job control, '**fg %ed**' would normally restart a suspended *ed*(1) job, if there were a suspended job whose name began with the string 'ed'.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a + and the previous job with a –. The abbreviation %+ refers to the current job and %– refers to the previous job. %% is also a synonym for the current job.

This shell learns immediately whenever a process changes state. 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 you try to leave the shell while jobs are running or stopped, you will be warned that `You have stopped(running) jobs.` You may use the *jobs* command to see what they are. 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 command are ignored if the command is followed by **&** and job **monitor** option is not active. Otherwise, signals have the values inherited by the shell from its parent (but see also the *trap* command).

Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the **Special Commands** listed below, it is executed within the current 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 issues a **return**, the positional parameter list is restored and any trap set on **EXIT** within the function is executed. The value of a *function* is the value of the last command executed. A function is also executed in the current shell process. If a command name is not a *special command* or a user defined *function*, a process is created and an attempt is made to execute the command via *exec*(2).

The shell parameter PATH defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is **/bin:/usr/bin:** (specifying **/bin**, **/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 /, 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 be a file containing shell commands. A sub-shell is spawned to read it. All non-exported aliases, functions, and named parameters are removed in this case. If the shell command file doesn't have read permission, or if the *setuid* and/or *setgid* bits are set on the file, then the shell executes an agent whose job it is to set up the permissions and execute the shell with the shell command file passed down as an open file. A parenthesized command is also executed in a sub-shell without removing non-exported quantities.

Command Re-entry

The text of the last HISTSIZE (default 128) commands entered from a terminal device is saved in a *history* file. The file **$HOME/.ksh_history** is used if the HISTFILE variable is not set or is not writable. A shell can access the commands of all *interactive* shells that use the same named **HISTFILE**. The special command *fc* is 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 can be specified. If you do not specify an editor program as an argument to *fc*, the value of the parameter **FCEDIT** is used. If **FCEDIT** is not defined, then **/bin/ed** is used. The edited command(s) is printed and re-executed upon leaving the editor. The editor name **–** is used to skip the editing phase and to re-execute the command. In this case, a substitution parameter of the form *old=new* can be used to modify the command before execution. For example, if **r** is aliased to **'fc –e –'** then typing **'r bad=good c'** will re-execute the most recent command which starts with the letter **c**, replacing the first occurrence of the string **bad** with the string **good**.

In-line Editing Options

Normally, each command line entered from a terminal device is simply typed followed by a newline ('RETURN' or 'LINE FEED'). If either the **emacs**, **gmacs**, or **vi** option is active, the user can edit the command line. To be in either of these edit modes **set** the corresponding option. An editing option is automatically selected each time the VISUAL or EDITOR variable is assigned a value ending in either of these option names.

The editing features require that the user's terminal accept 'RETURN' as carriage return without line feed and that a space (' ') must overwrite the current character on the screen.

The editing modes implement a concept where the user is looking through a window at the current line. The window width is the value of COLUMNS if it is defined, otherwise 80. If the line is longer than the window width minus two, a mark is displayed at the end of the window to notify the user. As the cursor moves and reaches the window boundaries the window will be centered about the cursor. The mark is a > (<, *) if the line extends on the right (left, both) side(s) of the window.

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 key. The SHIFT key is *not* depressed. (The notation ^? indicates the DEL 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 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 to character *char* on current line.

^X^X
Interchange the cursor and mark.

erase
(User defined erase character as defined by the *stty* 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), 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 lowercase.

^K
Kill from the cursor to the end of the line. If given a parameter of zero then kill from the start of line to the cursor.

^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 linefeed (useful when using paper terminals).

^Y
Restore last item removed from line. (Yank item back to the line.)

^L
Linefeed and print current line.

^@
(**NULL** character) Set mark.

M-
(Meta space) Set mark.

^J
(**NEWLINE**) Execute the current line.

^M
(**RETURN**) Execute the current line.

eof
EOF character, normally **^D**, will terminate the shell if the current line is **NULL**.

^P
Fetch previous command. Each time **^P** is entered the previous command back in time is accessed.

M-<
Fetch the least recent (oldest) history line.

M->
Fetch the most recent (youngest) history line.

^N

Fetch next command. Each time **^N** is entered the next command 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 omitted, 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*, **^D**, **^K**, **^R**, **^P**, **^N**, **M-.**, **M-_**, **M-b**, **M-c**, **M-d**, **M-f**, **M-h**, 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-.

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 does not contain any special pattern characters.

M-ESC

Same as **M-***.

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.

Vi Editing Mode

There are two typing modes. Initially, when you enter a command you are in the *input* mode. To edit, the user enters *control* mode by typing ESC (033) and moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. Most control commands accept an optional repeat *count* prior to the command.

When in *vi* mode on most systems, canonical processing is initially enabled and the command will be echoed again if the speed is 1200 baud or greater and it contains any control characters or less than one second has elapsed since the prompt was printed. The ESC character terminates canonical processing for the remainder of the command and the user can then modify the command line. This scheme has the advantages of canonical processing with the type-ahead echoing of raw mode.

If the option **viraw** is also set, the terminal will always have canonical processing disabled. This mode is implicit for systems that do not support two alternate end of line delimiters, and may be helpful for certain terminals.

Input Edit Commands

By default the editor is in input mode:

erase
(User defined erase character as defined by the *stty* command, usually ^H or #.) Delete previous character.

^W
Delete the previous blank separated word.

^D
Terminate the shell.

^V
Escape next character. Editing characters, the user's erase or kill characters may be entered in a command line or in a search string if preceded by a **^V**. The **^V** removes the next character's editing features (if any).

****
Escape the next *erase* or *kill* character.

Motion Edit Commands

These commands will move the cursor.

[*count*]**l**
Cursor forward (right) one character.

[*count*]**w**
Cursor forward one alpha-numeric word.

[*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 backward (left) one character.

[*count*]**b**
Cursor backward one word.

[*count*]**B**
Cursor to preceding blank separated word.

[*count*]**f***c*
Find the next character *c* in the current line.

[*count*]**F***c*
Find the previous character *c* in the current line.

[*count*]**t***c*
Equivalent to **f** followed by **h**.

[*count*]**T***c*
Equivalent to **F** followed by **l**.

;
Repeats the last single character find command, **f**, **F**, **t**, or **T**.

,
Reverses the last single character find command.

0
Cursor to start of line.

^

Cursor to first non-blank character in line.

$

Cursor to EOL.

Search Edit Commands

These commands access your command history:

[*count*]**k**

Fetch previous command. Each time **k** is entered the previous command back in time is accessed.

[*count*]**–**

Equivalent to **k**.

[*count*]**j**

Fetch next command. Each time **j** is entered the next command forward in time is accessed.

[*count*]**+**

Equivalent to **j**.

[*count*]**G**

The command number *count* is fetched. The default is the least recent history command.

/string

Search backward through history for a previous command containing *string*. *string* is terminated by a **RETURN** or **NEWLINE**. If *string* is **NULL** the previous string will be used.

?string

Same as / except that search will be in the forward direction.

n

Search for next match of the last pattern to / or **?** commands.

N

Search for next match of the last pattern to / or **?**, but in reverse direction. Search history for the *string* entered by the previous / command.

Text Modification Edit Commands

These commands will modify the line.

a

Enter input mode and enter text after the current character.

A

Append text to the end of the line. Equivalent to **$a**.

[*count*]**c***motion*
c[*count*]*motion*
Delete current character through the character that *motion* would move the cursor to and enter input mode. If *motion* is **c**, the entire line will be deleted and input mode entered.

C
Delete the current character through the end of line and enter input mode. Equivalent to **c$**.

S
Equivalent to **cc**.

D
Delete the current character through the EOL. Equivalent to **d$**.

[*count*]**d***motion*
d[*count*]*motion*
Delete current character through the character that *motion* would move to. If *motion* is **d**, the entire line will be deleted.

i
Enter input mode and insert text before the current character.

I
Insert text before the beginning of the line. Equivalent to the two character sequence **^i**.

[*count*]**P**
Place the previous text modification before the cursor.

[*count*]**p**
Place the previous text modification after the cursor.

R
Enter input mode and replace characters on the screen with characters you type in an overlay fashion.

r*c*
Replace the current character with *c*.

[*count*]**x**
Delete current character.

[*count*]**X**
Delete preceding character.

[*count*]**.**
Repeat the previous text modification command.

~
Invert the case of the current character and advance the cursor.

[*count*]_

Causes the *count* word of the previous command to be appended and input mode entered. The last word is used if *count* is omitted.

*

Causes an * to be appended to the current word and file name generation attempted. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.

Other Edit Commands

Miscellaneous commands:

[*count*]y*motion*
y[*count*]*motion*

Yank current character through character that *motion* would move the cursor to and puts them into the delete buffer. The text and cursor are unchanged.

Y

Yanks from current position to end of line. Equivalent to y$.

u

Undo the last text modifying command.

U

Undo all the text modifying commands performed on the line.

[*count*]v

Returns the command **fc –e ${VISUAL:–${EDITOR:–vi}}** *count* in the input buffer. If *count* is omitted, then the current line is used.

^L

Linefeed and print current line. Has effect only in control mode.

^J

(New line) Execute the current line, regardless of mode.

^M

(Return) Execute the current line, regardless of mode.

#

Sends the line after inserting a # in front of the line and after each newline. Useful for causing the current line to be inserted in the history without being executed.

=

List the file names that match the current word if an asterisk were appended it.

@letter

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 for processing.

Special Commands

The following simple-commands are executed in the shell process. Input/Output redirection is permitted. Unless otherwise indicated, the output is written on file descriptor 1. Commands that are preceded by one or two † are treated specially in the following ways:

1. Parameter assignment lists preceding the command remain in effect when the command completes.
2. They are executed in a separate process when used within command substitution.
3. Errors in commands preceded by †† cause the script that contains them to abort.

† **:** [*arg* . . .]

The command only expands parameters. A zero exit code is returned.

†† **.** *file* [*arg* . . .]

Read and execute commands from *file* and return. The commands are executed in the current 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.

alias [**–tx**] [*name* [=*value*] . . .]

Alias with no arguments prints the list of aliases in the form *name*=*value* on standard output. An *alias* is defined for each name whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution. The **–t** flag is 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 remained tracked. Without the **–t** flag, for each *name* in the argument list for which no *value* is given, the name and value of the alias is printed. The **–x** flag is used to set or print exported aliases. An exported alias is defined across sub-shell environments. Alias returns true unless a *name* is given for which no alias has been defined.

bg [*%job*]
This command is only built-in on systems that support job control. Puts the specified *job* into the background. The current job is put in the background if *job* is not specified.

break [*n*]
Exit from the enclosing **for, while, until,** or **select** loop, if any. If *n* is specified then break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for, while, until,** or **select** loop. If *n* is specified then resume at the *n*-th enclosing loop.

† **cd** [*arg*]
† **cd** *old new*
This command can be in either of two forms. In the first form it changes the current directory to *arg*. If *arg* is – the directory is changed to the previous directory. The shell parameter **HOME** is the default *arg*. The parameter **PWD** is set to the current directory. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a **NULL** pathname, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / then the search path is not used. Otherwise, each directory in the path is searched for *arg*.

The second form of **cd** substitutes the string *new* for the string *old* in the current directory name, **PWD** and tries to change to this new directory. The **cd** command may not be executed by *rksh*.

echo [*arg* . . .]
See *echo*(1) for usage and description.

†† **eval** [*arg* . . .]
The arguments are read as input to the shell and the resulting command(s) executed.

†† **exec** [*arg* . . .]

If *arg* is given, the command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and affect the current process. If no arguments are given, the effect of this command is to modify file descriptors as prescribed by the input/output redirection list. In this case, any file descriptor numbers greater than 2 that are opened with this mechanism are closed when invoking another program.

exit [*n*]

Causes the shell to exit with the exit status specified by *n*. If *n* is omitted then the exit status is that of the last command executed. An end-of-file will also cause the shell to exit except for a shell which has the *ignoreeof* option (see **set**) turned on.

†† **export** [*name* . . .]

The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands.

†† **fc** [**–e** *ename*] [**–nlr**] [*first*] [*last*]
†† **fc –e –** [*old=new*] [*command*]

In the first form, a range of commands from *first* to *last* is selected from the last **HISTSIZE** commands that were typed at the terminal. The arguments *first* and *last* may be specified as a number or as a string. A string is used to locate the most recent command starting with the given string. A negative number is used as an offset to the current command number.

If the flag **–l** is selected, the commands are listed on standard output. Otherwise, the editor program *ename* is invoked on a file containing these keyboard commands. If *ename* is not supplied, the value of the parameter **FCEDIT** (default **/bin/ed**) is used as the editor. When editing is complete, the edited command(s) is executed. If *last* is not specified, it will be set to *first*. If *first* is not specified the default is the previous command for editing and –16 for listing. The flag **–r** reverses the order of the commands and the flag **–n** suppresses command numbers when listing. In the second form, the *command* is re-executed after the substitution *old=new* is performed.

fexpr
Built-in (i.e., faster) version of *expr*(1). The same syntax applies. If the System V Interface Definition is ever extended to allow *expr* itself to be a special (built-in) command, *fexpr* may go away. To ensure upward compatibility, shells should be checked for the existence of *fexpr* before using it.

fg [*%job*]
This command is only built-in on systems that support job control. If *job* is specified it brings it to the foreground. Otherwise, the current job is brought into the foreground.

getops
Used in shell scripts to support command syntax standards (see *intro*(1)); it parses positional parameters and checks for legal options. See *getops*(1) for usage and description.

jobs [**–l**]
Lists the active jobs; given the **–l** options lists process id's in addition to the normal information.

kill [*–sig*] *process ...*
Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in **/usr/include/signal.h**, stripped of the prefix "SIG"). The signal numbers and names are listed by '**kill –l**'. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal if it is stopped. The argument *process* can be either a process id or a job.

let *arg ...*
Each *arg* is an *arithmetic expression* to be evaluated. All calculations are done as long integers and no check for overflow is performed. Expressions consist of constants, named parameters, and operators. The following set of operators, listed in order of decreasing precedence, have been implemented:

–	unary minus
!	logical negation
* / %	multiplication, division, remainder
+ –	addition, subtraction
<= >= < >	comparison
== !=	equality inequality
=	arithmetic replacement

Sub-expressions in parentheses **()** are evaluated first and can be used to override the above precedence rules. The evaluation within a precedence group is from right to left for the = operator and from left to right for the others.

A parameter name must be a valid *identifier*. When a parameter is encountered, the value associated with the parameter name is substituted and expression evaluation resumes. Up to nine levels of recursion are permitted.

The return code is 0 if the value of the last expression is non-zero, and 1 otherwise.

†† **newgrp** [*arg* . . .]
Equivalent to **exec newgrp** *arg* . . . .

print [**–Rnprsu**[*n*]] [*arg* . . .]
The shell output mechanism. With no flags or with flag **–**, the arguments are printed on standard output as described by *echo*(1). In raw mode, **–R** or **–r**, the escape conventions of *echo* are ignored. The **–R** option will print all subsequent arguments and options other than **–n**. The **–p** option causes the arguments to be written onto the pipe of the process spawned with **|&** instead of standard output. The **–s** option causes the arguments to be written onto the history file instead of standard output. The **–u** flag can be used to specify a one digit file descriptor unit number *n* on which the output will be placed. The default is 1. If the flag **–n** is used, no **new-line** is added to the output.

pwd
Equivalent to **print –r – $PWD**

read [**–prsu** [*n*]] [*name***?***prompt*] [*name* . . .]
The shell input mechanism. One line is read and is broken up into words using the characters in **IFS** as separators. In raw mode, **–r**, a **** at the end of a line does not signify line continuation. The first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*.

The **–p** option causes the input line to be taken from the input pipe of a process spawned by the shell using **|&**. If the **–s** flag is present, the input will be saved as a command in the history file. The flag **–u** can be used to specify a one digit file descriptor unit to read from. The file descriptor can be opened with the **exec** special command. The default value of *n* is 0.

If *name* is omitted, **REPLY** is used as the default *name*. The return code is 0 unless an EOF is encountered. An EOF with the **–p** option causes cleanup for this process so that another can be spawned. If the first argument contains a **?**, the remainder of this word is used as a *prompt* when the shell is interactive. If the given file descriptor is open for writing and is a terminal device, the prompt is placed on this unit. Otherwise, the prompt is issued on file descriptor 2. The return code is 0 unless an EOF is encountered.

†† **readonly** [*name* . . .]
The given *names* are marked readonly and these names cannot be changed by subsequent assignment.

†† **return** [*n*]
Causes a shell *function* to return to the invoking script with the return status specified by *n*. If *n* is omitted then the return status is that of the last command executed. If **return** is invoked while not in a *function* or a **.** script, then it is the same as an **exit**.

set [**–aefhkmnostuvx**] [**–o** *option* . . .] [*arg* . . .]
The flags for this command have meaning as follows:

–a
All subsequent parameters that are defined are automatically exported.

–e
If the shell is non-interactive and if a command fails, execute the **ERR** trap, if set, and exit immediately. This mode is disabled while reading profiles.

–f
Disables file name generation.

–h
Each command whose name is an *identifier* becomes a tracked alias when first encountered.

–k
All parameter assignment arguments are placed in the environment for a command, not just those that precede the command name.

–m
Background jobs will run in a separate process group and a line will print upon completion. The exit status of background jobs is reported in a completion message. On systems with job control, this flag is turned on automatically for interactive shells.

–n
Read commands but do not execute them. Ignored for interactive shells.

–o
The following argument can be one of the following option names:

allexport	Same as **–a**.
errexit	Same as **–e**.
bgnice	All background jobs are run at a lower priority.
emacs	Puts you in an *emacs* style in-line editor for command entry.
gmacs	Puts you in a *gmacs* style in-line editor for command entry.
ignoreeof	The shell will not exit on end-of-file. The command **exit** must be used.
keyword	Same as **–k**.
markdirs	All directory names resulting from file name generation have a trailing / appended.
monitor	Same as **–m**.
noexec	Same as **–n**.
noglob	Same as **–f**.
nounset	Same as **–u**.
protected	Same as **–p**.
verbose	Same as **–v**.
trackall	Same as **–h**.
vi	Puts you in insert mode of a *vi* style in-line editor until you hit escape character **033**. This puts you in move mode. A return sends the line.

viraw Each character is processed as it is typed in *vi* mode.
xtrace Same as **–x**.
If no option name is supplied then the current option settings are printed.

–p
Resets the PATH variable to the default value, disables processing of the **$HOME/.profile** file and uses the file **/etc/suid_profile** instead of the **ENV** file. This mode is automatically enabled whenever the effective uid (gid) is not equal to the real uid (gid).

–s
Sort the positional parameters.

–t
Exit after reading and executing one command.

–u
Treat unset parameters as an error when substituting.

–v
Print shell input lines as they are read.

–x
Print commands and their arguments as they are executed.

–
Turns off **–x and –v** flags and stops examining arguments for flags.

––
Do not change any of the flags; useful in setting **$1** to a value beginning with –. If no arguments follow this flag then the positional parameters are unset.

Using + rather than – causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in **$–**. The remaining arguments are positional parameters and are assigned, in order, to **$1 $2 ....** If no arguments are given then the values of all names are printed on the standard output.

† **shift** [*n*]
The positional parameters from **$***n***+1** ... are renamed **$1** ... , default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to **$#**.

test [*expr*]
Evaluate conditional expression *expr*. See *test*(1) for usage and description. The arithmetic comparison operators are not restricted to integers. They allow any arithmetic expression. Four additional primitive expressions are allowed:

–L *file*	True if *file* is a symbolic link.
file1 **–nt** *file2*	True if *file1* is newer than *file2*.
file1 **–ot** *file2*	True if *file1* is older than *file2*.
file1 **–ef** *file2*	True if *file1* has the same device and i-node number as *file2*.

times
Print the accumulated user and system times for the shell and for processes run from the shell.

trap [*arg*] [*sig*] ...
arg is a command to be read and executed when the shell receives signal(s) *sig*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Each *sig* can be given as a number or as the name of the signal.

Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If *arg* is omitted or is –, then all trap(s) *sig* are reset to their original values. If *arg* is the **NULL** string then this signal is ignored by the shell and by the commands it invokes. If *sig* is **ERR** then *arg* will be executed whenever a command has a non-zero exit code. This trap is not inherited by functions. If *sig* is **0** or **EXIT** the *trap* statement is executed inside the body of a function, then the command *arg* is executed after the function completes. If *sig* is 0 or **EXIT** for a *trap* set outside any function then the command *arg* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

†† **typeset** [**–HLRZfilprtux**[*n*] [*name* [=*value*]] ...]
When invoked inside a function, a new instance of the parameter *name* is created. The parameter value and type are restored when the function completes. The following list of attributes may be specified:

–H
This flag provides SYSTEM V/88 to host-name file mapping on non-SYSTEM V/88 machines.

–L

Left justify and remove leading blanks from *value*. If *n* is non-zero, it defines the width of the field; otherwise, it is determined by the width of the value of first assignment. When the parameter is assigned, it is filled on the right with blanks or truncated, if necessary, to fit into the field. Leading zeros are removed if the **–Z** flag is also set. The **–R** flag is turned off.

–R

Right justify and fill with leading blanks. If *n* is non-zero, it defines the width of the field; otherwise, it is determined by the width of the value of first assignment. The field is left filled with blanks or truncated from the end if the parameter is reassigned. The **L** flag is turned off.

–Z

Right justify and fill with leading zeros if the first non-blank character is a digit and the **–L** flag has not been set. If *n* is non-zero, it defines the width of the field; otherwise, it is determined by the width of the value of first assignment.

–f

The names refer to function names rather than parameter names. No assignments can be made and the only other valid flags are **–t**, which turns on execution tracing for this function and **–x**, to allow the function to remain in effect across shell procedures executed in the same process environment.

–i

Parameter is an integer. This makes arithmetic faster. If *n* is non-zero, it defines the output arithmetic base; otherwise, the first assignment determines the output base.

–l

All uppercase characters converted to lowercase. The uppercase flag, **–u** is turned off.

–p

The output of this command, if any, is written onto the two-way pipe.

–r

The given *names* are marked readonly and these names cannot be changed by subsequent assignment.

–t
Tags the named parameters. Tags are user definable and have no special meaning to the shell.

–u
All lowercase characters are converted to uppercase characters. The lowercase flag, **–l** is turned off.

–x
The given *names* are marked for automatic export to the *environment* of subsequently-executed commands.

Using + instead of – causes these flags to be turned off. If no *name* arguments are given but flags are specified, a list of *names* (and optionally the *values* of the *parameters* which have these flags set is printed. (Using + rather than – keeps the values to be printed.) If no *names* and flags are given, the *names* and *attributes* of all *parameters* are printed.

ulimit [–acdfmpst] [*n*]

–a
lists all of the current resource limits (BSD (Berkeley System Development) only).

–c
imposes a size limit of *n* 512 byte blocks on the size of core dumps (BSD only).

–d
imposes a size limit of *n* kbytes on the size of the data area (BSD only).

–f
imposes a size limit of *n* 512 byte blocks on files written by child processes (files of any size may be read).

–m
imposes a soft limit of *n* kbytes on the size of physical memory (BSD only).

–p
changes the pipe size to *n* (UNIX/RT only).

–s
imposes a size limit of *n* kbytes on the size of the stack area (BSD only).

–t
imposes a time limit of *n* seconds to be used by each process (BSD only).

If no option is given, **–f** is assumed. If *n* is not given the current limit is printed.

umask [*nnn*]
The user file-creation mask is set to *nnn* (see *umask*(2)). If *nnn* is omitted, the current value of the mask is printed.

unalias *name* ...
The parameters given by the list of *names* are removed from the *alias* list.

unset [–f] *name* ...
The parameters given by the list of *names* are unassigned, i.e., their values and attributes are erased. Readonly variables cannot be unset. If the flag, **–f**, is set, then the names refer to *function* names.

wait [*n*]
Wait for the specified child process and report its termination status. If *n* is not given, then all currently active child processes are waited for. The return code from this command is that of the process waited for.

whence [–v] *name* ...
For each *name*, indicate how it would be interpreted if used as a command name. The flag, –v, produces a more verbose report.

Invocation

If the shell is invoked by *exec*(2), and the first character of argument zero ($0) is –, the shell is assumed to be a *login* shell and commands are read from **/etc/profile**, then from either **.profile** in the current directory or **$HOME/.profile**, if either file exists. Next, commands are read from the file named by performing parameter substitution on the value of the environment parameter **ENV** if the file exists. If the **–s** flag is not present and *arg* is, then a path search is performed on the first *arg* to determine the name of the script to execute. The script *arg* must have read permission and any *setuid* and *getgid* settings will be ignored. Commands are then read as described below; the following flags are interpreted by the shell when it is invoked:

–c *string*
If the –c flag is present then commands are read from *string*.

–s
If the –s flag is present or if no arguments remain then commands are read from the standard input. Shell output, except for the output of the **Special Commands** listed above, is written to file descriptor 2.

–i

If the **–i** flag is present or if the shell input and output are attached to a terminal (as told by *ioctl*(2)) then this shell is *interactive*. 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

If the **–r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

rksh Only

If you wish to use *ksh* as the shell, *sh*, and as the restricted shell, *rsh*, copy *ksh* into *sh* and link *rsh* to *sh*. Otherwise, you may call *ksh* as it is and link *rksh* to *ksh*.

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 *ksh*, 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*.

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. Otherwise, the shell returns the exit status of the last command executed (see also the *exit* command above). If the shell is being used non-interactively then execution of the shell file is abandoned. 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, then the line number is also printed in square brackets ([ ]) after the command or function name.

FILES

/etc/passwd
/etc/profile
/etc/suid_profile
$HOME/.profile
/tmp/ksh*
/dev/null

SEE ALSO

cat(1), cd(1), echo(1), emacs(1), env(1), gmacs(1), newgrp(1), test(1), umask(1), vi(1), dup(2), exec(2), fork(2), ioctl(2), lseek(2), pipe(2), signal(2), umask(2), ulimit(2), wait(2), rand(3), a.out(5), profile(5), environ(7).

CAVEATS

If a command that is a *tracked alias* is executed, then a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the **–t** option of the *alias* command to correct this situation.

Some very old shell scripts contain a ^ as a synonym for the pipe character, |.

If a command is piped into a shell command, all variables set in the shell command are lost when the command completes.

Using the **fc** built-in command within a compound command will cause the whole command to disappear from the history file.

The built-in command **.** *file* reads the whole file before any commands are executed. Therefore, *alias* and *unalias* commands in the file will not apply to any functions defined in the file.

NAME

ld – link editor for common object files

SYNOPSIS

ld [*options*] *filename*

DESCRIPTION

The *ld* command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and *ld* combines the objects, producing an object module that can either be executed or, if the **–r** option is specified, used as input for a subsequent *ld* run. The output of *ld* is left in **a.out**. By default this file is executable if no errors occurred during the load. If any input file, *filename*, is not an object file, *ld* assumes it is either an archive library or a text file containing link editor directives. (See *Link Editor Directives* in the *Programmer's Guide* for a discussion of input directives.)

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table (see **ar**(4)) is searched sequentially with as many passes as are necessary to resolve external references which can be satisfied by library members. Thus, the ordering of library members is functionally unimportant, unless there exist multiple library members defining the same external symbol.

The following options are recognized by **ld**:

–e *epsym*
Set the default entry point address for the output file to be that of the symbol **epsym**.

–f *fill*
Set the default fill pattern for "holes" within an output section as well as initialized *bss* sections. The argument *fill* is a 2-byte constant.

–l*x*
Search a library **lib***x***.a**, where *x* is up to nine characters. A library is searched when its name is encountered, so the placement of a –l is significant. By default, libraries are located in **LIBDIR** or **LLIBDIR**.

–m
Produce a map or listing of the input/output sections on the standard output.

–o *outfile*
Produce an output object file by the name **outfile**. The name of the default object file is **a.out**.

–r
Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent *ld* run. The link editor will not complain about unresolved references, and the output file will not be executable.

–a
Create an absolute file. This is the default if the option is not used. Used with the **–r** option, **–a** allocates memory for common symbols.

–s
Strip line number entries and symbol table information from the output object file.

–t
Turn off the warning about multiply-defined symbols that are not the same size.

–u *symname*
Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the *ld* line is significant; it must be placed before the library which will define the symbol.

–x
Do not preserve local symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

–z
Do not bind anything to address zero. This option will allow runtime detection of **NULL** pointers.

–L *dir*
Change the algorithm of searching for **lib*x*.a** to look in **dir** before looking in **LIBDIR** and **LLIBDIR**. This option is effective only if it precedes the **–l** option on the command line.

–M
Output a message for each multiply-defined external definition.

–N
Put the text section at the beginning of the text segment rather than after all header information, and put the data section immediately following text in the core image.

–V
Output a message giving information about the version of ld being used.

–VS *num*
Use **num** as a decimal version stamp identifying the **a.out** file that is produced. The version stamp is stored in the optional header.

–Y*[LU],dir*
Change the default directory used for finding libraries. If **L** is specified, the first default directory that *ld* searches, **LIBDIR**, is replaced by *dir*. If **U** is specified and *ld* has been built with a second default directory, **LLIBDIR**, then that directory is replaced by *dir*. If *ld* was built with only one default directory and **U** is specified, a warning is printed and the option is ignored.

FILES

LIBDIR/libx.a	libraries
LLIBDIR/libx.a	libraries
a.out	output file
LIBDIR	usually /lib
LLIBDIR	usually /usr/lib

SEE ALSO

as(1), cc(1), exit(2), end(3C), a.out(4), ar(4), and *Link Editor Directives* in the *Programmer's Guide.*

CAVEATS

Through its options and input directives, the common link editor gives users great flexibility; however, those who use the input directives must assume some added responsibilities. Input directives and options should ensure the following properties for programs: C defines a zero pointer as NULL. A pointer to which zero has been assigned must not point to any object. To satisfy this, users must not place any object at virtual address zero in the program's address space. When the link editor is called through cc(1), a startup routine is linked with the user's program. This routine calls *exit*(~) (see *exit* (2)) after execution of the main program. If the user calls the link editor directly, then the user must ensure that the program always calls *exit*(~) instead of falling through the end of the entry routine.

The symbols *etext*, *edata*, and *end* (see *end*(3C)) are reserved and are defined by the link editor. It is incorrect for a user program to redefine them.

If the link editor does not recognize an input file as an object file or an archive file, it will assume that it contains link editor directives and will attempt to parse it. This will occasionally produce an error message complaining about "syntax errors".

Arithmetic expressions may only have one forward referenced symbol per expression.

Shared libraries are not supported.

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [**–rctvn**] [*file*] ...

DESCRIPTION

The *lex* command generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file.

The strings may contain square brackets to indicate character classes, as in **[abx–z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators ***, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrence of, the previous character or character class. The character **.** is the class of all ASCII characters except newline. Parentheses for grouping and vertical bar for alternation are also supported.

The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation. Thus, **[a–zA–Z]+** matches a string of letters. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character **$** at the end of an expression requires a trailing newline. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \.

Three subroutines defined as macros are expected: *input*() to read a character; *unput*(c) to replace a character read; and *output*(c) to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named *yylex*() , and the library contains a *main*() which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function *yymore*() accumulates additional characters into the same *yytext*; and the function *yyless*(p) pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext+yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes **%%** it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a **%%**, as in YACC. Lines preceding **%%** which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with **{ }**. Note that curly brackets do not imply parentheses; only string substitution is done.

EXAMPLE

```
D           [0–9]
%%
if          printf("IF statement\n");
[a–z]+      printf("tag, value %s\n",yytext);
0{D}+       printf("octal number %s\n",yytext);
{D}+        printf("decimal number %s\n",yytext);
"++"        printf("unary op\n");
"+"         printf("binary op\n");
"/*"          skipcommnts();
%%
 skipcommnts()
 {
          for (;;)
          {
                    while (input() != '*')
                              ;
                    if (input() != '/')
                              unput(yytext[yyleng-1]);
                    else
                              return;
          }
 }
```

The external names generated by *lex* all begin with the prefix yy or YY.

The flags must appear before any files. The flag **–r** indicates RATFOR actions, **–c** indicates C actions and is the default, **–t** causes the lex.yy.c program to be written instead to standard output, **–v** provides a one-line summary of statistics, **–n** will not print out the **–v** summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default 2500)

%n *n* number of states is *n* (500)

%e *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions is *n* (2000)

%k *n* number of packed character classes is *n* (1000)

%o *n* size of output array is *n* (3000)

The use of one or more of the above automatically implies the **–v** option, unless the **–n** option is used.

SEE ALSO

yacc(1)
Programmer's Guide.

BUGS

The **–r** option is not yet fully operational.

NAME

line – read one line

SYNOPSIS

line

DESCRIPTION

line copies one line (up to a newline) from the standard input and writes it on the standard output. It returns an exit code of 1 on **EOF** and always prints at least a newline. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1).
read(2) in the *Programmer's Reference Manual.*

NAME

lint – a C program checker

SYNOPSIS

lint [*option*] ... *file* ...

DESCRIPTION

The *lint* command attempts to detect features of the C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with **.c** are taken to be C source files. Arguments whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the **–c** or the **–o** option used. The **.ln** files are analogous to **.o** (object) files that are produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

lint will take all the **.c**, **.ln**, and **llib-l***x***.ln** (specified by **–l***x*) files and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the **–p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the **–c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the **–c** option is used, the **.ln** and the **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

–a
Suppress complaints about assignments of long values to variables that are not long.

–b
Suppress complaints about **`break`** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in many such complaints).

–h
Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

–u
Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program).

–v
Suppress complaints about unused arguments in functions.

–x
Do not report variables referred to by external declarations but never used.

The following arguments alter *lint*'s behavior:

–l*x*
Include additional lint library **llib-l***x***.ln**. For example, you can include a lint version of the math library **llib-lm.ln** by inserting –lm on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

–n
Do not check compatibility against either the standard or the portable lint library.

–p
Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

–c
Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lints* first pass only and are not checked for inter-function compatibility.

–o lib
Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **–c** option nullifies any use of the **–o** option. The lint library produced is the input that is given to *lints* second pass. The **–o** option simply causes this file to be saved in the named lint library.

To produce a **llib-l*lib*.ln** without extraneous messages, use of the **–x** option is suggested. The **–v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of "lint comments" (see below).

The **–D**, **–U**, and **–I** options of *cpp*(1) and the **–g** and **–O** options of *cc*(1) are also recognized as separate arguments. The **–g** and **–O** options are ignored, but, by recognizing these options, *lints* behavior is closer to that of the *cc*(1) command. Other options are warned about and ignored. The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions like *exit*(2)).

/*VARARGS n */
suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
turns on the **–v** option for the next function.

/*LINTLIBRARY*/
at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **–v** and **–x** options.

lint produces its first output on a per-source-file basis. Complaints about included files are collected and printed after all source files have been processed. Finally, if the **–c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the **–c** and the **–o** options allows for incremental use of *lint* on a set of C source files. Generally, you invoke *lint* once for each source file with the **–c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **–c** option), listing all the **.ln** files with the needed **–l***x* options. This will print all the inter-file inconsistencies. This scheme works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were *lint*ed.

FILES

LLIBDIR	the directory where the lint libraries specified by the **–l***x* option must exist, usually /usr/lib
LLIBDIR/lint[12]	first and second passes
LLIBDIR/llib-lc.ln	declarations for C Library functions (binary format; source is in *LLIBDIR*/llib-lc)
LLIBDIR/llib-port.ln	declarations for portable functions (binary format; source is in *LLIBDIR*/llib-port)
LLIBDIR/llib-lm.ln	declarations for Math Library functions (binary format; source is in *LLIBDIR*/llib-lm)
TMPDIR/*lint*	temporaries
TMPDIR	usually /usr/tmp but can be redefined by setting the environment variable **TMPDIR** (see *tempnam*() in *tmpnam*(3S)).

SEE ALSO

cc(1), cpp(1), make(1).

BUGS

exit(2), *setjmp*(3C), and other functions that do not return are not understood; this causes various lies.

NAME

list – from a common object file produce a C source listing with line numbers

SYNOPSIS

list [–V] [–h] [-F *function*] source file [*source file* . . .] [*object-file*]

DESCRIPTION

The **list** command produces a C source listing with line number information attached. If multiple C source files were used to create the object file, **list** will accept multiple file names. The object file is taken to be the last non-C source file argument. If no object file is specified, the default object file, **a.out**, will be used.

Line numbers will be printed for each line marked as breakpoint inserted by the compiler (generally, each executable C statement that begins a new line of source). Line numbering begins anew for each function. Line number 1 is always the line containing the left curly brace ({) that begins the function body. Line numbers will also be supplied for inner block redeclarations of local variables so that they can be distinguished by the symbolic debugger.

The following options are interpreted by **list** and may be given in any order:

–V
Print, on standard error, the version number of the *list* command executing.

–h
Suppress heading output.

–F*function*
List only the named *function.* The **–F** option may be specified multiple times on the command line.

SEE ALSO

as(1), cc(1), ld(1).

CAVEATS

Object files given to *list* must have been compiled with the **–g** option of cc(1).

Since *list* does not use the C preprocessor, it may be unable to recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

DIAGNOSTICS

If *name* cannot be read, *list* will produce the error message:

```
list: name: cannot open
```

If the source file names do not end in **.c** , the message is:

```
list: name: invalid C source name.
```

An invalid object file name will cause the message:

```
list: name: bad magic
```

If any of the symbolic debugging information is missing, one of the following messages will be printed:

```
list: name: symbols have been stripped, cannot
            proceed,
list: name: cannot read line numbers
list: name: not in symbol table.
```

The following messages are produced when *list* becomes confused by **#ifdefs** in the source file:

```
list: name: cannot find function in symbol table
list: name: out of sync: too many }
list: name: unexpected end-of-file.
```

When either symbol debugging information is missing, or *list* has been confused by C preprocessor statements. it displays the error message:

```
list: name: missing or inappropriate line
numbers
```

NAME

locate – identify a command using keywords

SYNOPSIS

[help] locate
[help] locate [*keyword1* [*keyword2*] ...]

DESCRIPTION

The *locate* command is part of the Help Facility, and provides online assistance with identifying commands.

Without arguments, the initial *locate* screen is displayed from which the user may enter keywords functionally related to the action of the commands they wish to have identified. A user may enter keywords and receive a list of commands whose functional attributes match those in the keyword list, or may exit to the shell by typing **q** (for "quit"). For example, if you wish to print the contents of a file, enter the keywords "print" and "file". The *locate* command would then print the names of all commands related to these keywords.

Keywords may also be entered directly from the shell, as shown above. In this case, the initial screen is not displayed, and the resulting command list is printed.

More detailed information on a command in the list produced by *locate* can be obtained by accessing the *usage* module of the Help Facility. Access is made by entering the appropriate menu choice after the command list is displayed.

From any screen in the Help Facility, a user may execute a command via the shell (*sh*(1)) by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable **SCROLL** must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your *.profile* file (see *profile*(4)): "export SCROLL ; SCROLL=no". If you later decide that scrolling is desired, **SCROLL** must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

SEE ALSO

glossary(1), help(1), sh(1), starter(1), usage(1)
term(5) in the *Programmer's Reference Manual.*

WARNINGS

If the shell variable TERM (see *sh*(1)) is not set in the user's **.profile** file, then TERM will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

login – sign on

SYNOPSIS

login [*name* [*env-var* ...]]

DESCRIPTION

The *login* command is used at the beginning of each terminal session and allows you to identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a **CTRL-D** to indicate an EOF.

If *login* is invoked as a command, it must replace the initial command interpreter. This is accomplished by typing the following from the initial shell:

exec login

login asks for your user name (if not supplied as an argument) and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it will not appear on the written record of the session.

If you make any mistake in the login procedure, you will receive the message:

```
Login incorrect
```

A new login prompt will appear. If you make five incorrect login attempts, all five may be logged in **/usr/adm/loginlog** (if it exists) and the line will be dropped.

If you do not complete the login successfully within a certain period of time (e.g., one minute), you are likely to be silently disconnected.

After a successful login, accounting files are updated, the procedure **/etc/profile** is performed, the message-of-the-day, if any, is printed, the user-ID, the group-ID, the working directory, and the command interpreter (usually **sh**(1)) is initialized, and the file **.profile** in the working directory is executed, if it exists. These specifications are found in the **/etc/passwd** file entry for the user. The name of the command interpreter is **–** followed by the last component of the interpreter's pathname (i.e., **–sh**). If this field in the password file is empty, then the default command interpreter, **/bin/sh** is used. If this field is *, then the named directory becomes the root directory, the starting point for path searches for pathnames beginning with a /. At that point, *login* is re-executed at the new level which must have its own root structure, including **/etc/login** and **/etc/passwd.**

The basic environment is initialized to:

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
TZ=*timezone-specification*

The environment may be expanded or modified by supplying additional arguments to *login*, either at execution time or when *login* requests your login name. The arguments may take either the form *xxx* or *xxx=yyy*. Arguments without an equal sign are placed in the environment as:

L*n*=*xxx*

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variables containing an = are placed into the environment without modification. If they already appear in the environment, they replace the older value. There are two exceptions. The variables PATH and SHELL cannot be changed. This prevents people, logging into restricted shell environments, from spawning secondary shells that are not restricted. Both *login* and *getty* understand simple single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

FILES

/etc/utmp	accounting
/etc/wtmp	accounting
/usr/mail/*your-name*	**mailbox for user** *your-name*
/usr/adm/loginlog	record of failed login attempts
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/shadow	shadow password file
/etc/profile	system profile
.profile	user's login profile

SEE ALSO

mail(1), newgrp(1), sh(1), su(1M)
loginlog(4), passwd(4), profile(4), environ(5) in the *System Administrator's Reference Manual*.

DIAGNOSTICS

login incorrect if the user name or the password cannot be matched.
No shell, cannot open password file, or no directory: consult a UNIX system programming counselor.

No utmp entry. You must exec *login* from the lowest level **sh** if you attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

NAME

logname – get login name

SYNOPSIS

logname

DESCRIPTION

logname returns the contents of the environment variable $LOGNAME which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1)
logname(3X), environ(5) in the *Programmer's Reference Manual.*

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder *file* ...

DESCRIPTION

The input is one or more object or library archive *files* (see *ar*(1)). The standard output is a list of pairs of object file or archive member names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1). Note that the link editor *ld*(1) is capable of multiple passes over an archive in the portable archive format (see *ar*(4)) and does not require that *lorder*(1) be used when building an archive. The usage of the *lorder*(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing **.o** files:

```
ar –cr library `lorder *.o | tsort`
```

FILES

TMPDIR/*symref	temporary files
TMPDIR/*symdef	temporary files

TMPDIR is usually **/usr/tmp** but can be redefined by setting the environment variable TMPDIR (see *tempnam*() in *tmpnam*(3S)).

SEE ALSO

ar(1), ld(1), tsort(1), ar(4)

CAVEAT

lorder will accept as input any object or archive file, regardless of its suffix, provided there is more than one input file. If there is but a single input file, its suffix must be **.o**.

NAME

lp, cancel – send/cancel requests to an LP print service

SYNOPSIS

lp [*printing-options*] *files*

lp –i *request-ids printing-options*

cancel [*request-ids*] [*printers*]

DESCRIPTION

The first form of the *lp* shell command arranges for the named files and associated information (collectively called a *request*) to be printed. If no file names are specified on the shell command line, the standard input is assumed. The standard input may be specified along with named *files* on the shell command line by using the file name(s) and – for the standard input. The *files* will be printed in the order they appear on the shell command line.

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).

lp associates a unique *request-id* with each request and prints it on the standard output. This *request-id* can be used later to cancel, change, or find the status of the request. (See the section on *cancel* for details about canceling a request, the previous paragraph for an explanation of how to change a request, and *lpstat*(1) for information about checking the status of a print request.)

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 listed in any order. The following options are available for *lp*:

–c

Make copies of the *files* to be printed immediately when *lp* is invoked. Normally, *files* will not be copied. 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

in the absence of the **–c** option, 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*
Print this request using *dest* as the printer or class of printers. Under certain conditions (lack of printer availability, capabilities of printers, and so on), requests for specific destinations may not be accepted (see *accept*(1M) and *lpstat*(1)). By default, *dest* is taken from the environment variable **LPDEST** (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary from system to system (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 any 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 **hold**, **resume**, and **immediate**, as defined below:

hold
Do not 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 of the output. (Default is 1.)

–o *option*
Specify printer-dependent or class-dependent *options*. Several such options may be specified on a single command line either by using the **–o** keyletter more than once (i.e., **–o** $option_1$ **–o** $option_2$ ... **–o** $option_n$), or by specifying a list of options with one **–o** keyletter (i.e., **–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 cannot 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 cannot 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 cannot 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 columns per inch) or **elite** (representing 12 columns per inch), or it can be **compressed** (representing as many columns as a printer can handle). There is no standard number of columns per inch for all printers; see the Terminfo database (*terminfo*(4)) for the default character pitch for your printer.

This option cannot be used with the **–f** option.

stty=*stty-option-list*
A list of options valid for the *stty* command; enclose the list with 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.

–s
Suppress messages from the print service such as request id is *request-id*.

–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.

–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, but 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 can be used only if there is a filter available to handle it; otherwise, the print request will be rejected.

Canceling a Print Request

The *cancel* command cancels printer requests that were made by the *lp*(1) shell command. The shell command line arguments may be either *request-ids* (as returned by *lp*(1)) or *printer* names (for a complete list, use *lpstat*(1)). Specifying a *request-id* cancels the associated request even if it is currently printing. Specifying a *printer* cancels the request that is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

NOTES

Printers for which requests are not being accepted will not be considered when the destination is **any**. (Use the **lpstat –a** command to see which printers are accepting requests.) On the other hand, if a request is destined for a class of printers and the class itself is accepting requests, *all* printers in the class will be considered, regardless of their acceptance status, as long as the printer class is accepting requests.

WARNING

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 do not use the **–S** option.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lpstat(1), mail(1).
accept(1M), lpadmin(1M), lpfilter(1M), lpforms(1M), lpsched(1M), lpusers(1M) in the *System Administrator's Reference Manual.*
terminfo(4) in the *Programmer's Reference Manual.*

NAME

lpstat – print information about the status of the LP Print Service

SYNOPSIS

lpstat [*options*]

DESCRIPTION

lpstat prints information about the current status of the LP Print Service.

If no options are given, then *lpstat* prints the status of all requests made to *lp*(1) by users. Any arguments that are not options are assumed to be *request-ids* (as returned by *lp*), printers, or printer classes. *lpstat* prints the status of such requests, printers, or printer classes. 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 enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

–u "user1, user2, user3"

Specifying "all" after any keyletters that take "list" as an argument causes all information relevant to the keyletter to be printed. For example, the following command prints the status of all output requests.

lpstat –o all

The options are:

–a [*list*]
Display acceptance status (with respect to *lp*) of destinations for requests (see *accept*(1M)). *list* is a list of intermixed printer names and class names; the default is **all**.

–c [*list*]
Display class names and their members. *list* is a list of class names; the default is **all**.

–d
Display the system default destination for *lp*.

–f [*list*] [**–l**]
Display 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*] [**–l**]
Display the status of output requests. *list* is a list of intermixed printer names, class names, and request-ids; the default is **all**. The **–l** option gives a more detailed status of the request.

–p [*list*] [**–D**] [**–l**]
Display the status of printers named in *list*. *list* is a list of printers; the default is **all**. If the **–D** option is given, a brief description is printed for each printer in *list*. If the **–l** option is given, 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.

–r
Display the status of the LP request scheduler.

–s
Display a status summary, including the system default destination, a list of class names and their members, a list of printers and their associated devices, a list of all forms currently mounted, and a list of all recognized character sets and print wheels.

–S [*list*] [**–l**]
Display 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
Display all status information.

–u [*list*]
Display the status of output requests for users. *list* is a list of login names; the default is **all**.

–v [*list*]
Display the names of printers and the path names of the devices associated with them. *list* is a list of printer names; the default is **all**.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1).

NAME

ls – list contents of directory

SYNOPSIS

ls [**–RadLCxmlnogrtucpFbqisf**] [*names*]

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The default format is to list one entry per line, the **–C** and **–x** options enable multi-column formats, and the **–m** option enables stream output format. In order to determine output formats for the **–C**, **–x**, and **–m** options, *ls* uses an environment variable, COLUMNS, to determine the number of character positions available on one output line. If this variable is not set, the *terminfo*(4) database is used to determine the number of columns, based on the environment variable TERM. If this information cannot be obtained, 80 columns are assumed.

The *ls* command has the following options:

–R
Recursively list subdirectories encountered.

–a
List all entries, including those that begin with a dot (.), which are normally not listed.

–d
If an argument is a directory, list only its name (not its contents); often used with **–l** to get the status of a directory.

–L
If an argument is a symbolic link, list the file or directory the link references instead of the link itself.

–C
Multi-column output with entries sorted down the columns.

–x
Multi-column output with entries sorted across instead of down the page.

–m
Stream output format; files are listed across the page, separated by commas.

–l
List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will instead contain the major and minor device numbers instead of a size.

–n
The same as **–l**, except that the owner's **UID** and group's GID numbers are printed, instead of the associated character strings.

–o
The same as **–l**, except that the group is not printed.

–g
The same as **–l**, except that the owner is not printed.

–r
Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

–t
Sort by time stamp (latest first) instead of by name. The default is the last modification time. (See **–n** and **–c**.)

–u
Use time of last access instead of last modification for sorting (with the **–t** option) or printing (with the **–l** option).

–c
Use time of last modification of the i-node (file created, mode changed, etc.) for sorting (**–t**) or printing (**–l**).

–p
Put a slash (/) after each filename if that file is a directory.

–F
Put a slash (/) after each filename if that file is a directory, an asterisk (*) after each filename if that file is executable, and an at-sign (@) after each filename if that file is a symbolic link.

–b
Force printing of non-printable characters to be in the octal **\ddd** notation.

–q
Force printing of non-printable characters in file names as the character question mark (**?**).

–i
For each file, print the i-number in the first column of the report.

–s
Give size in blocks, including indirect blocks, for each entry.

–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.

The mode printed under the **–l** option consists of ten characters. The first character may be one of the following:

d	the entry is a directory
b	the entry is a block special file
c	the entry is a character special file
l	the entry is a symbolic link
p	the entry is a fifo (a.k.a. "named pipe") special file
–	the entry is an ordinary file

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the file; and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

ls –l (the long list) prints output as:

```
-rwxrwxrwx  1 smith  dev     10876  May 16 9:42 part2
```

This horizontal configuration provides a good deal of information. Reading from right to left, you see that the current directory holds one file, named "part2." Next, the last time that file's contents were modified was 9:42 A.M. on May 16. The file is moderately sized, containing 10,876 characters, or bytes. The owner of the file, or the user, belongs

to the group "dev" (perhaps indicating "development"), and his or her login name is "smith." The number, in this case "1," indicates the number of links to file "part2." Finally, the row of dash and letters tell you that user, group, and others have permissions to read, write, execute "part2."

The execute (x) symbol here occupies the third position of the three-character sequence. A − in the third position would have indicated a denial of execution permissions.

The permissions are indicated as:

- **r** the file is readable
- **w** the file is writable
- **x** the file is executable
- **–** the indicated permission is *not* granted
- **l** mandatory locking will occur during access (the set-group-ID bit is on and the group execution bit is off)
- **s** the set-user-ID or set-group-ID bit is on, and the corresponding user or group execution bit is also on
- **S** undefined bit-state (the set-user-ID bit is on and the user execution bit is off)
- **t** the 1000 (octal) bit, or sticky bit, is on (see *chmod*(1)), and execution is on
- **T** the 1000 bit is turned on, and execution is off (undefined bit-state)

For user and group permissions, the third position is sometimes occupied by a character other than x or −. **s** also may occupy this position, referring to the state of the set-ID bit, whether it be the user's or the group's. The ability to assume the same ID as the user during execution is, for example, used during login when you begin as root but need to assume the identity of the user stated at "login."

In the case of the sequence of group permissions, **l** may occupy the third position. **l** refers to mandatory file and record locking. This permission describes a file's ability to allow other files to lock its reading or writing permissions during access.

For others permissions, the third position may be occupied by **t** or T. These refer to the state of the sticky bit and execution permissions.

EXAMPLES

An example of a file's permissions is:

```
-rwxr--r--
```

This describes a file that is readable, writable, and executable by the user and readable by the group and others.

Another example of a file's permissions is:

```
-rwsr-xr-x
```

This describes a file that is readable, writable, and executable by the user, readable and executable by the group and others, and allows its user-ID to be assumed, during execution, by the user presently executing it.

Another example of a file's permissions is:

```
-rw-rwl---
```

This describes a file that is readable and writable only by the user and the group and can be locked during access.

An example of a command line:

ls –a

This command will print the names of all files in the current directory, including those that begin with a dot (.), which normally do not print.

Another example of a command line:

ls –aisn

This command will provide you with quite a bit of information including all files, including non-printing ones (**a**), the **i**-number—the memory address of the i-node associated with the file—printed in the left-hand column (**i**); the **s**ize (in blocks) of the files, printed in the column to the right of the i-numbers (**s**); finally, the report is displayed in the **n**umeric version of the long list, printing the UID (instead of user name) and GID (instead of group name) numbers associated with the files.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

FILES

/etc/passwd	user IDs for **ls –l** and **ls –o**
/etc/group	group IDs for **ls –l** and **ls –g**
/usr/lib/terminfo/?/*	terminal information database

SEE ALSO

chmod(1), find(1).

NOTES

In an RFS environment, you may not have the permissions that the output of the *ls –l* command leads you to believe. For more information, see the *Mapping Remote Users* section of Chapter 10 of the *System Administrator's Guide*.

BUGS

Unprintable characters in file names may confuse the columnar output options.

NAME

m4 – macro processor

SYNOPSIS

m4 [*options*] [*files*]

DESCRIPTION

The *m4* command is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is –, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

–e
Operate interactively. Interrupts are ignored and the output is unbuffered.

–s
Enable line sync output for the C preprocessor (#line . . .)

–B*int*
Change the size of the push-back and argument collection buffers from the default of 4,096.

–H*int*
Change the size of the symbol table hash array from the default of 199. The size should be prime.

–S*int*
Change the size of the call stack from the default of 100 slots. Macros take three slots, and non-macro arguments take one.

–T*int*
Change the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any **–D** or **–U** flags:

–D*name*[=*val*]
Defines *name* to *val* or to `NULL` in *val*'s absence.

–U*name*
undefines *name*.

Macro calls have the form:

name(arg1,arg2, . . ., argn)

The (must immediately follow the name of the macro. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be NULL. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

m4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are NULL unless otherwise stated.

define
: the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the NULL string; $# is replaced by the number of arguments; $* is replaced by a list of all the arguments separated by commas; $@ is like $*, but each argument is quoted (with the current quotes).

undefine
: removes the definition of the macro named in its argument.

defn
: returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef
: like *define*, but saves any previous definition.

popdef
removes current definition of its argument(s), exposing the previous one, if any.

ifdef
if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is NULL. The word *unix* is predefined on UNIX system versions of *m4*.

shift
returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote
change quote symbols to the first and second arguments. The symbols may be up to five characters long. *Changequote* without arguments restores the original values (i.e., ` ').

changecom
change left and right comment markers from the default # and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert
m4 maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert
causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum
returns the value of the current output stream.

dnl
reads and discards characters up to and including the next newline.

ifelse
has three or more arguments. If the first argument is the same string as the second, the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, NULL.

incr
returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr
returns the value of its argument decremented by 1.

eval
evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, –, *, /, %, ^ (exponentiation), bitwise &, |, ^, and ~; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len
returns the number of characters in its argument.

index
returns the position in its first argument where the second argument begins (zero origin), or –1 if the second argument does not occur.

substr
returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit
transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include
returns the contents of the file named in the argument.

sinclude
is identical to *include*, except that it says nothing if the file is inaccessible.

syscmd
executes the command given in the first argument. No value is returned.

sysval
is the return code from the last call to *syscmd*.

maketemp
fills in a string of XXXXX in its argument with the current process ID.

m4exit
causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.

m4wrap
argument 1 will be pushed back at final EOF; example: m4wrap(`cleanup()')

errprint
prints its argument on the diagnostic output file.

dumpdef
prints current names and definitions, for the named items, or for all if no arguments are given.

traceon
with no arguments, turns on tracing for all macros (including builtins). Otherwise, turns on tracing for named macros.

traceoff
turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

SEE ALSO
cc(1), cpp(1).

NAME

machid: m68k, m88k, pdp11, u3b, u3b2, u3b5, vax – get processor type truth value

SYNOPSIS

m68k

m88k

pdp11

u3b

u3b2

u3b5

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.

m68k
True if you are on a Motorola M68000-Family-based microcomputer.

m88k
True if you are on a Motorola M88000-Family-based microcomputer.

pdp11
True if you are on a PDP-11/45 or PDP-11/70.

u3b
True if you are on a 3B20 computer.

u3b2
True if you are on a 3B2 computer.

u3b5
True if you are on a 3B5 computer.

vax
True if you are on a VAX-11/750 or VAX-11/780.

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

sh(1), test(1), true(1).
make(1) in the *Programmer's Reference Manual*.

NAME

mail, rmail – send mail to users or read mail

SYNOPSIS

Sending mail:

mail [**–oswt**] *persons*

rmail [**–oswt**] *persons*

Reading mail:

mail [**–ehpqr**] [**–f** *file*] [**–F** *persons*]

DESCRIPTION

Sending mail:

The command-line arguments that follow affect SENDING mail:

–o
suppresses the address optimization facility.

–s
suppresses the addition of a **NEWLINE** at the top of the letter being sent. See WARNINGS below.

–w
causes a letter to be sent to a remote user without waiting for the completion of the remote transfer program.

–t
causes a **To:** line to be added to the letter, showing the intended recipients.

A *person* is usually a user name recognized by *login*(1). When *persons* are named, *mail* assumes a message is being sent (except in the case of the **–F** option). It reads from the standard input up to an end-of-file (control-d), or until it reads a line consisting of just a period. When either of those signals is received, *mail* adds the *letter* to the *mailfile* for each *person*. A *letter* is a *message* preceded by a *postmark*. The message is preceded by the sender's name and a *postmark*. A *postmark* consists of one or more 'From' lines followed by a blank line (unless the –s argument was used).

If a letter is found to be undeliverable, it is returned to the sender with diagnostics that indicate the location and nature of the failure. If *mail* is interrupted during input, the file **dead.letter** is saved to allow editing and resending. **dead.letter** is recreated every time it is needed, erasing any previous contents.

rmail only permits the sending of mail; *uucp*(1C) uses *rmail* as a security precaution.

If the local system has the BNU installed, mail may be sent to a recipient on a remote system. Prefix *person* by the system name and exclamation point. A series of system names separated by exclamation points can be used to direct a letter through an extended network.

Reading Mail:

The command-line arguments that follow affect READING mail:

–e
causes mail not to be printed. An exit value of 0 is returned if the user has mail; otherwise, an exit value of 1 is returned.

–h
causes a window of headers to be displayed rather than the latest message. The display is followed by the '**?**' prompt.

–p
causes all messages to be printed without prompting for disposition.

–q
causes *mail* to terminate after interrupts. Normally an interrupt causes only the termination of the message being printed.

–r
causes messages to be printed in first-in, first-out order.

–f*file*
causes *mail* to use *file* (e.g., **mbox**) instead of the default *mailfile*.

–F*persons*
entered into an empty *mailbox*, causes all incoming mail to be forwarded to *persons*.

mail, unless otherwise influenced by command-line arguments, prints a user's mail messages in last-in, first-out order. For each message, the user is prompted with a **?**, and a line is read from the standard input. The following commands are available to determine the disposition of the message:

newline, **+**, or **n**
Go on to next message.

d, or **dp**
Delete message and go on to next message.

d #
Delete message number **#**. Do not go on to next message.

dq
Delete message and quit *mail*.

h
Display a window of headers around current message.

h #
Display header of message number #.

h a
Display headers of ALL messages in the user's *mailfile*.

h d
Display headers of messages scheduled for deletion.

p
Print current message again.

–
Print previous message.

a
Print message that arrived during the *mail* session.

#
Print message number #.

r [*users* **]**
Reply to the sender, and other *user(s)*, then delete the message.

s [*files*]
Save message in the named *files* (**mbox** is default).

y
Same as save.

u [#]
Undelete message number # (default is last read).

w [*files*]
Save message, without its top-most header, in the named *files* (**mbox** is default).

m [*persons*]
Mail the message to the named *persons*.

q, or **ctl-d**
Put undeleted mail back in the *mailfile* and quit *mail*.

x
Put all mail back in the *mailfile* unchanged and exit *mail*.

!*command*
Escape to the shell to do *command*.

?
Print a command summary.

When a user logs in, the presence of mail, if any, is indicated. Also, notification is made if new mail arrives while using *mail*.

The *mailfile* may be manipulated in two ways to alter the function of *mail*. The *other* permissions of the file may be read-write, read-only, or neither read nor write to allow different levels of privacy. If changed to other than the default, the file will be preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

Forward to *person*

which will cause all mail sent to the owner of the *mailfile* to be forwarded to *person*. A "Forwarded by..." message is added to the header. This is especially useful in a multi-machine environment to forward all of a person's mail to a single machine, and to keep the recipient informed if the mail has been forwarded. Installation and removal of forwarding is done with the **–F** option.

To forward all of your mail to systema!user, type:

mail -Fsystema!user

To forward to more than one user, type:

mail -F"user1,systema!user2,systema!systemb!user3"

Note that when more than one user is specified, the whole list should be enclosed in double quotes so that it may all be interpreted as the operand of the **–F** option. The list can be up to 1024 bytes; either commas or white space can be used to separate users.

To remove forwarding, type:

mail –F ""

The pair of double quotes is mandatory to set a `NULL` argument for the **–F** option.

In order for forwarding to work properly the *mailfile* should have "mail" as group ID, and the group permission should be read-write.

FILES

/etc/passwd	to identify sender and locate persons
/usr/mail/user	incoming mail for *user*; i.e., the *mailfile*
$HOME/mbox	saved mail
$MAIL	variable containing pathname of *mailfile*
/tmp/ma*	temporary file
/usr/mail/*.lock	lock for mail directory
dead.letter	unmailable text

SEE ALSO

login(1), mailx(1), write(1).
User's Guide.
System Administrator's Guide.

WARNING

The "Forward to person" feature may result in a loop, if *sys1!userb* forwards to *sys2!userb* and *sys2!userb* forwards to *sys1!userb*. The symptom is a message saying "unbounded...saved mail in dead.letter."

The **–s** option should be used with caution. It allows the text of a message to be interpreted as part of the postmark of the letter, possibly causing confusion to other *mail* programs. To allow compatibility with *mailx*(1), if the first line of the message is "Subject:...", the addition of a <newline> is suppressed whether or not the **–s** option is used.

BUGS

Conditions sometimes result in a failure to remove a lock file.

After an interrupt, the next message may not be printed; printing may be forced by typing a p.

NAME

mailx – interactive message processing system

SYNOPSIS

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

DESCRIPTION

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

Many of the remote features of *mailx* will only work if BNU are installed on your system.

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

The user can access a secondary file by using the **–f** option of the *mailx* command. Messages in the secondary file can then be read or otherwise processed using the same COMMANDS as in the primary *mailbox*. This gives rise within these pages to the notion of a current *mailbox*.

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

–e
Test for presence of mail. *mailx* prints nothing and exits with a successful return code if there is mail to read.

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

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

–h *number*
The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops. (See *addsopt* under **ENVIRONMENT VARIABLES.**)

–H
Print header summary only.

–i
Ignore interrupts. (See *ignore* under **ENVIRONMENT VARIABLES.**)

–n
Do not initialize from the system default *mailx.rc* file.

–N
Do not print initial header summary.

–r *address*
Pass *address* to network delivery software. All tilde commands are disabled. (See *addsopt* under **ENVIRONMENT VARIABLES.**)

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

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

–U
Convert *uucp* style addresses to internet standards. Overrides the "conv" environment variable. (See *addsopt* under **ENVIRONMENT VARIABLES.**)

When reading mail, *mailx* is in *command mode*. A header summary of the first several messages is displayed, followed by a prompt indicating *mailx* can accept regular commands (see **COMMANDS**). When sending mail, *mailx* is in *input mode*. If no subject is specified on the command line, a prompt for the subject is printed. (A "subject" longer than 1024 characters will cause *mailx* to dump core) As the message is typed, *mailx* will read the message and store it in a temporary file. Commands may be entered by beginning a line with the tilde (~) escape character followed by a single command letter and optional arguments. See **TILDE ESCAPES** for a summary of these commands.

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

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

Regular commands are of the form

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

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

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

n	Message number **n**.
.	The current message.
^	The first undeleted message.
$	The last message.
*****	All messages.
n–m	An inclusive range of message numbers.
user	All messages from **user**.

/string All messages with **string** in the subject line (case ignored).

:*c* All messages of type *c*, where *c* is one of:

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

Note that the context of the command determines whether this type of message specification makes sense.

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

At start-up time, *mailx* tries to execute commands from the optional system-wide file (**/usr/lib/mailx/mailx.rc**) to initialize certain parameters, then from a private start-up file (**$HOME/.mailrc**) for personalized variables. With the exceptions noted below, regular commands are legal inside start-up files. The most common use of a start-up file is to set up initial display options and alias lists. The following commands are not legal in the start-up file: **!**, **Copy**, **edit**, **followup**, **Followup**, **hold**, **mail**, **preserve**, **reply**, **Reply**, **shell**, and **visual**. An error in the start-up file causes the remaining lines in the file to be ignored. The **.mailrc** file is optional, and must be constructed locally.

COMMANDS

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

!*shell-command*
Escape to the shell. See *SHELL* (**ENVIRONMENT VARIABLES**).

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

=
Print the current message number.

?
Prints a summary of commands.

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

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

cd [*directory*]
chdir [*directory*]
Change directory. If *directory* is not specified, $HOME is used.

copy [*filename*]
copy [*msglist*] *filename*
Copy messages to the file without marking the messages as saved. Otherwise, equivalent to the save command.

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

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

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

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

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

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

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

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

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

Default file is the current *mailbox*.

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

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

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

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

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

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

help
Prints a summary of commands.

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

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

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

list
Prints all commands available. No explanation is given.

mail *name ...*
Mail a message to the specified users.

Mail *name*
Mail a message to the specified user and record a copy of it in a file named after that user.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

version
Prints the current version and release date.

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

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

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

z[+ | –]
Scroll the header display forward or backward one screen–full. The number of headers displayed is set by the *screen* variable (see **ENVIRONMENT VARIABLES**).

TILDE ESCAPES

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

~ ! *shell-command*
Escape to the shell.

~ .
Simulate end of file (terminate message input).

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

~ ?
Print a summary of tilde escapes.

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

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

~ b *name* ...
Add the *name*s to the blind carbon copy (Bcc) list.

~ c *name* ...
Add the *name*s to the carbon copy (Cc) list.

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

~ e
Invoke the editor on the partial message. See also *EDITOR* (**ENVIRONMENT VARIABLES**).

~ f [*msglist*]
Forward the specified messages. The messages are inserted into the message without alteration.

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

~ i *string*
Insert the value of the named variable into the text of the message. For example, **~A** is equivalent to '**~i** Sign.' Environment variables set and exported in the shell are also accessible by ~i.

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

~ p
Print the message being entered.

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

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

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

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

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

~ w *filename*
Write the partial message onto the given file, without the header.

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

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

ENVIRONMENT VARIABLES

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

HOME=*directory*
The user's base of operations.

MAILRC=*filename*
The name of the start-up file. Default is **$HOME/.mailrc**.

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

addsopt
Enabled by default. If **/bin/mail** is not being used as the deliverer, **noaddsopt** should be specified. (See **WARNING**.)

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

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

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

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

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

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

cmd=*shell-command*
Set the default command for the **pipe** command. No default value.

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

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

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

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

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

EDITOR=*shell-command*
The command to run when the **edit** or **~e** command is used. Default is *ed*(1).

escape=*c*
Substitute *c* for the ~ escape character. Takes effect with next message sent.

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

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

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

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

ignoreeof
Ignore end-of-file during message input. Input must be terminated by a period (.) on a line by itself or by the ~. command. Default is **noignoreeof**. See also *dot* above.

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

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

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

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

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

onehop
When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FILES

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

SEE ALSO
ls(1), mail(1), pg(1).

WARNINGS
The **–h, –r** and **–U** options can be used only if *mailx* is built with a delivery program other than **/bin/mail.**

BUGS

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

Internal variables imported from the execution environment cannot be unset.

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

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

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [**–f** makefile] [**–p**] [**–i**] [**–k**] [**–s**] [**–r**] [**–n**] [**–b**] [**–e**] [**–u**] [**–t**] [**–q**] [*names*]

DESCRIPTION

The *make* command allows the programmer to maintain, update, and regenerate groups of computer programs. The following is a brief description of all options and some special names:

–f *makefile*
Description filename. *makefile* is assumed to be the name of a description file.

–p
Print out the complete set of macro definitions and target descriptions.

–i
Ignore error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

–k
Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

–s
Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

–r
Do not use the built-in rules.

–n
No execute mode. Print commands, but do not execute them. Even lines beginning with an @ are printed.

–b
Compatibility mode for old makefiles.

–e
Environment variables override assignments within makefiles.

–t
Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

–q
Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

.DEFAULT
If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS
Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT
Same effect as the **–s** option.

.IGNORE
Same effect as the **–i** option.

make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **–f** option is present, **makefile**, **Makefile**, and the SCCS files **s.makefile**, and **s.Makefile** are tried in order. If *makefile* is **–**, the standard input is taken. More than one **–f** *makefile* argument pair may appear.

make updates a target only if its dependents are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out-of-date.

makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-**NULL** list of targets, then a **:**, then a (possibly **NULL**) list of prerequisite files or dependencies. Text following a **;** and all following lines that begin with a tab are shell commands to be executed to update the target. The first non-empty line that does not begin with a tab or **#** begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><newline> sequence. Everything printed by *make* (except the initial tab) is passed directly to the shell as is.

Thus,

```
echo a\
b
```

will produce

```
ab
```

exactly the same as the shell would.

Sharp (**#**) and new-line surround comments.

The following *makefile* says that **pgm** depends on two files **a.o** and b.o, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h**:

```
pgm: a.o b.o
        cc a.o b.o –o pgm
a.o: incl.h a.c
        cc –c a.c
b.o: incl.h b.c
        cc –c b.c
```

Command lines are executed one at a time, each by its own shell. The **SHELL** environment variable can be used to specify which shell *make* should use to execute commands. The default is **/bin/sh**. The first one or two characters in a command can be the following: **–**, **@**, **–@**, or **@–**. If **@** is present, printing of the command is suppressed. If **–** is present, *make* ignores an error. A line is printed when it is executed unless the **–s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the initial character sequence contains a **@**. The **–n** option specifies printing without execution; however, if the command line has the string **$(MAKE)** in it, the line is always executed (see discussion of the **MAKEFLAGS** macro under *Environment*). The **–t** (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate *make*. If the **–i** option is present, or the entry **.IGNORE:** appears in *makefile*, or the initial character sequence of the command contains **–**, the error is ignored. If the **–k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The **–b** option allows old makefiles (those written for the old version of *make*) to run without errors.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the special name **.PRECIOUS**.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The **–e** option causes the environment to override the macro assignments in a makefile. Suffixes and their associated rules in the makefile will override any identical suffixes in the built-in rules.

The **MAKEFLAGS** environment variable is processed by *make* as containing any legal input option (except **–f** and **–p**) defined for the command line. Further, upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the **–n** option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make –n** recursively on a whole software system to see what would have been executed. This is because the **–n** is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Include Files

If the string *include* appears as the first seven letters of a line in a *makefile*, and is followed by a blank or a tab, the rest of the line is assumed to be a filename and will be read by the current invocation, after substituting for any macros.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped new-line. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional :*subst1*=*subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, new-line characters, and beginnings of lines. An example of the use of the substitute sequence is shown under **Libraries**.

Internal Macros

There are five internally maintained macros that are useful for writing rules for building targets.

$*

The macro **$*** stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@

The **$@** macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$<

The **$<** macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module that is out-of-date with respect to the target (i.e., the "manufactured" dependent file name). Thus, in the **.c.o** rule, the **$<** macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc –c –O $*.c
```

or:

```
.c.o:
        cc –c –O $<
```

$?

The **$?** macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out-of-date with respect to the target; essentially, those modules which must be rebuilt.

$%

The **$%** macro is only evaluated when the target is an archive library member of the form **lib(file.o)**. In this case, **$@** evaluates to **lib** and **$%** evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros, the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part, **./** is generated. The only macro excluded from this alternative form is **$?**.

Suffixes

Certain names (for instance, those ending with **.o**) have inferable prerequisites such as **.c**, **.s**, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c .c~ .f .f~ .sh .sh~
.c.o .c.a .c~.o .c~.c .c~.a
.f.o .f.a .f~.o .f~.f .f~.a
.h~.h .s.o .s~.o .s~.s .s~.a .sh~.sh
.l.o .l.c .l~.o .l~.l .l~.c
.y.o .y.c .y~.o .y~.y .y~.c
```

The internal rules for *make* are contained in the source file **rules.c** for the *make* program. These rules can be locally modified. To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

make –fp – 2>/dev/null </dev/null

A tilde in the above rules refers to an SCCS file (see *sccsfile*(4)). Thus, the rule **.c~.o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix, it is incompatible with *make*'s suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e., **.c:**) is the definition of how to build *x* from *x***.c**. In effect, the other suffix is **NULL**. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~ .f .f~

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o –o pgm
a.o b.o: incl.h
```

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS, LFLAGS,** and **YFLAGS** are used for compiler options to *cc*(1), *lex*(1), and *yacc*(1), respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with **.c.o:** as the target and no dependents. Shell commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library that contains **file.o**. (This assumes the **LIB** macro has been previously defined.)The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form **.*XX*.a** where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(file.o)** depend upon **file.o** explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up-to-date
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

In fact, the **.c.a** rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) lib $?
        rm $?  @echo lib is now up-to-date
.c.a:;
```

Here, the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object filenames (inside **lib**) whose C source files are out-of-date. The substitution mode translates the **.o** to **.c**. (Unfortunately, you cannot as yet transform to **.c~**; however, this may become possible in the future.)Note also, the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile and s.[Mm]akefile
/bin/sh

SEE ALSO

cc(1), lex(1), yacc(1), printf(3S), sccsfile(4).
cd(1), sh(1) in the *User's Reference Manual*.

NOTES

Some commands return non-zero status inappropriately; use **-i** to overcome the difficulty.

BUGS

Filenames with the characters = : @ will not work. Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*. The syntax **(lib(file1.o file2.o file3.o)** is illegal. You cannot build **lib(file.o)** from **file.o**. The macro **$(a:.o=.c~)** does not work. Named pipes are not handled well.

NAME

makekey – generate encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It attempts to read 8 bytes for its *key* (the first eight input bytes), then it attempts to read 2 bytes for its *salt* (the last two input bytes). The output depends on the input in a way intended to be difficult to compute (i.e., to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, ., /, and upper- and lower-case letters. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4,096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but broken in 4,096 different ways. Using the *input key* as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 *output key* bits in the result.

makekey is intended for programs that perform encryption. Usually, its input and output will be pipes.

SEE ALSO

ed(1), crypt(1), vi(1).
passwd(4) in the *System Administrator's Reference Manual.*

CAVEATS

makekey can produce different results depending upon whether the input is typed at the terminal or redirected from a file.

WARNING

This command is provided with the Security Administration Utilities, which is only available in the United States.

NAME

man – display entries from this manual

SYNOPSIS

man [*options*] [*section*] *title* ...

DESCRIPTION

The *man* program locates and prints each entry of this manual named *title* in the specified *section*. (For historical reasons, the word "page" is often used as a synonym for "entry" in this context.) The *title* is entered in lowercase characters. The *section* number may have a letter suffix. When *section* is omitted, searches the whole manual for *title* and prints all occurrences of it. The *options* and their meanings are:

–T*term*
Print the entry as appropriate for terminal type *term*. For a list of recognized values of *term*, type **help term2**. The default value of *term* is **450**.

–w
Print on the standard output only the pathnames of the entries, relative to **/usr/catman**, or to the current directory for **–d** option.

–d
Search the current directory rather than **/usr/catman**; requires the full file name (e.g., **cu.1c**, rather than just **cu**).

–c
Invoke **col**(1); note that **col**(1) is invoked automatically by **man** unless *term* is one of **300, 300s, 450, 37, 4000a, 382, 4014, tek, 1620,** and X.

The *man* program examines the environment variable $TERM (see *environ*(5)) and attempts to select options that adapt the output to the terminal being used. The **–T***term* option overrides the value of $TERM; this may be used when sending the output of *man* to a line printer.

Section may be changed before each *title*.

As an example: man man

would reproduce on the terminal this entry, as well as any other entries named **man** that may exist in other sections of the manual, e.g., *man*(5).

FILES

/usr/catman/?_man/man[1-8][a-z]?/* Preformatted manual entries

SEE ALSO

environ(5), term(5).

CAUTION

The *man* command prints manual entries that were formatted by nroff when your operating system was installed. Entries are originally formatted with terminal type 37 and are printed using the correct terminal filters as derived from the –T*term* and $TERM settings.

NAME

mcs – manipulate the object file comment section

SYNOPSIS

mcs [*options*] *object-files*

DESCRIPTION

The *mcs* command manipulates the comment section in an object file. It is used to add to, delete, print, and compress the contents of the **.comment** section in a UNIX object file. If the object file is an archive, the file is treated as a set of individual object files. As a result, if the **–a** option is specified, the string is appended to the comment section of each archive element.

The following options are available.

–a *string*
Appends *string* to the comment section of the object files.

–c
Compresses the contents of the comment section. All duplicate entries are removed. The order of the remaining entries is not disturbed.

–d
Deletes the contents of the comment section from the object file. Also removes the object-file comment-section header.

–n *name*
Specifies the name of the section to access. By default, *mcs* deals with the one named **.comment**. This option can be used to specify another section.

–p
Prints the contents of the comment section on the standard output. When more than one name is specified, tags each printed entry with the name of the file from which it was extracted.

–P
Prints the contents of the comment section, tagging each line with the name of the file from which it was extracted. The format used is "filename:string."

EXAMPLES

mcs -p file	# Print *file*'s *comment* section.
mcs -a string file	# Append *string* to *file*'s *comment* section

SEE ALSO

cpp(1), a.out(4).

NAME

mesg – permit or deny messages

SYNOPSIS

mesg [**–n**] [**–y**]

DESCRIPTION

mesg with argument **n** forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *mesg* with argument **y** reinstates permission. All by itself, *mesg* reports the current state without changing it.

FILES

/dev/tty*

SEE ALSO

write(1).

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

mkdir – make directories

SYNOPSIS

mkdir [**–m** *mode*] [**–p**] *dirname* ...

DESCRIPTION

mkdir creates the named directories in mode 777 (possibly altered by *umask*(1)).

Standard entries in a directory (e.g., the files **.**, for the directory itself, and **..**, for its parent) are made automatically. *mkdir* cannot create these entries by name. Creation of a directory requires write permission in the parent directory.

The owner ID and group ID of the new directories are set to the process's effective user ID and group ID, respectively.

Two options apply to *mkdir*:

–m
This option allows users to specify the mode to be used for new directories. Choices for modes can be found in *chmod*(1).

–p
With this option, *mkdir* creates *dirname* by creating all the non-existing parent directories first.

EXAMPLE

To create the subdirectory structure **ltr/jd/jan**, type:

mkdir -p ltr/jd/jan

SEE ALSO

sh(1), rm(1), umask(1).
intro(2), mkdir(2) in the *Programmer's Reference Manual*.

DIAGNOSTICS

mkdir returns exit code 0 if all directories given in the command line were made successfully. Otherwise, it prints a diagnostic and returns non-zero. An error code is stored in *errno*.

NAME

mnt, umnt – mount and dismount file system

SYNOPSIS

mnt [*options*] [*name* [*directory*]]

umnt [*options*] [*name*]

DESCRIPTION

mnt announces to the system that a removable file system is present on the special device. The *directory* must exist already; it becomes the name of the root of the newly mounted file system.

umnt announces to the system that the removable file system previously mounted on a special device is to be removed.

mnt (umnt) has an optional argument, *name*. This argument is used to search the **permissions** file to determine the real device to mount (unmount). The file is searched and when *name* matches either the *slice* or the *alias* entry on a line, the *slice* entry is then used as the special device to be mounted (unmounted).

If no name is specified by the options or by the *name* argument, then the alias **floppy** is used.

By convention *mount*(1M) and *umount*(1M) require root permission to execute. Normal users must use *mnt/umnt* when dealing with mountable media.

The options available are:

–*c*

Mount (unmount) all file systems in the set *c*. Acceptable values for *c* are **A, a, B, b, 1, 2,** or **3**. Set membership is specified in the "perms" field of *permissions* file entries.

–d *dev [dir]*

Mount (unmount) *dev* as *dir* if given, otherwise as *mnt_pt* given in permissions file.

–f *[dir]*

Mount (unmount) the device **floppy** as *dir* if given, otherwise as *mnt_pt* given in permissions file.

—l*list_of_devs*
Mount (unmount) all names listed as the corresponding *mnt_pt*s given in permissions file.

–r
Mount the device read-only.

FILES

/etc/mnttab	mount table
/etc/mount	
/etc/umount	
/etc/filesys	permissions file

SEE ALSO

mount(1M) in the *System Administrator's Reference Manual.*
mnttab(4), filesys(4) in the *Programmer's Reference Manual.*

DIAGNOSTICS

mnt issues a warning if the file system to be mounted is currently mounted under another name.

umnt complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or a user's working directory.

BUGS

Some degree of validation is done on the file system; however, it is generally unwise to mount garbage file systems.

NAME

mt – magnetic tape control

SYNOPSIS

mt [**–f** *tapename*] *command* [*count*]

DESCRIPTION

The *mt* program sends commands to a magnetic tape drive. If *tapename* is not specified, the environment variable TAPE is used; if TAPE does not exist, *mt* uses the device **/dev/rmt/ctapen**.

NOTE: *tapename* must refer to a raw (not block) tape device.

By default *mt* performs the requested operation once. Operations may be performed repeatedly by specifying a *count* argument. Some of the operations may not be supported on cartridge media.

The available commands are listed below. Only as many characters as are required to uniquely identify a command need be specified.

mt returns a 0 exit status when the operation(s) were successful, 1 if the command was unrecognized, and 2 if an operation failed.

OPTIONS

eof, weof
Write *count* EOF marks at the current position on the tape.

fsf
Forward space *count* files.

bsf
Backwards space *count* files.

fsr
Forward space *count* records.

bsr
Backwards space *count* records.

For the following commands, *count* is ignored:

rewind
Rewind the tape.

retension
Wind the tape to the end of the reel and then rewind it, smoothing out the tape tension. (*count* is ignored.)

erase
Erase the entire tape.

FILES

/dev/rmt/*	raw magnetic tape interface

SEE ALSO

environ(5V).

NAME

newform – change the format of a text file

SYNOPSIS

newform [**–s**] [**–i***tabspec*] [**–o***tabspec*] [**–b***n*] [**–e***n*] [**–p***n*] [**–a***n*] [**–f**] [**–c***char*] [**–l***n*] [files]

DESCRIPTION

newform reads lines from the named *files*, or the standard input if no input file is named, and reproduces the lines on the standard output. Lines are reformatted in accordance with command line options in effect.

Except for **–s**, command line options may appear in any order, may be repeated, and may be intermingled with the optional *files*. Command line options are processed in the order specified. This means that option sequences like "**–e**15 **–l**60" will yield results different from "**–l**60 **–e**15". Options are applied to all *files* on the command line.

–s

Shears off leading characters on each line up to the first tab and places up to 8 of the sheared characters at the end of the line. If more than 8 characters (not counting the first tab) are sheared, the eighth character is replaced by a * and any characters to the right of it are discarded. The first tab is always discarded.

An error message and program exit will occur if this option is used on a file without a tab on each line. The characters sheared off are saved internally until all other options specified are applied to that line. The characters are then added at the end of the processed line.

For example, to convert a file with leading digits, one or more tabs, and text on each line, to a file beginning with the text, all tabs after the first expanded to spaces, padded with spaces out to column 72 (or truncated to column 72), and the leading digits placed starting at column 73, the command would be:

newform –s –i –l –a –e file-name

–i*tabspec*

Input tab specification: expands tabs to spaces, according to the tab specifications given. *Tabspec* recognizes all tab specification forms described in *tabs*(1). In addition, *tabspec* may be —, in which *newform* assumes that the tab specification is to be found in the first line read from the standard input (see *fspec*(4)). If no *tabspec* is given, *tabspec* defaults to **–8**. A *tabspec* of **–0** expects no tabs; if any are found, they are treated as **–1**.

–o*tabspec*

Output tab specification: replaces spaces by tabs, according to the tab specifications given. The tab specifications are the same as for **–i***tabspec*. If no *tabspec* is given, *tabspec* defaults to **–8**. A *tabspec* of **–0** means that no spaces will be converted to tabs on output.

–b*n*

Truncate *n* characters from the beginning of the line when the line length is greater than the effective line length (see **–l***n*). Default is to truncate the number of characters necessary to obtain the effective line length. The default value is used when **–b** with no *n* is used. This option can be used to delete the sequence numbers from a COBOL program as follows:

newform –l1 –b7 file-name

–e*n*

Same as **–b***n* except that characters are truncated from the end of the line.

–p*n*

Prefix *n* characters (see **–c***k*) to the beginning of a line when the line length is less than the effective line length. Default is to prefix the number of characters necessary to obtain the effective line length.

–a*n*

Same as **–p***n* except characters are appended to the end of a line.

–f

Write the tab specification format line on the standard output before any other lines are output. The tab specification format line which is printed will correspond to the format specified in the *last* **–o** option. If no **–o** option is specified, the line which is printed will contain the default specification of **–8**.

–c*k*

Change the prefix/append character to *k*. Default character for *k* is a space.

–l*n*

Set the effective line length to *n* characters. If *n* is not entered, **–l** defaults to 72. The default line length without the **–l** option is 80 characters. Note that tabs and backspaces are considered to be one character (use **–i** to expand tabs to spaces).

The **–l1** must be used to set the effective line length shorter than any existing line in the file so that the **–b** option is activated.

DIAGNOSTICS

All diagnostics are fatal.

usage: . . .
newform was called with a bad option.

not –s format
There was no tab on one line.

can't open file
Self-explanatory.

internal line too long
A line exceeds 512 characters after being expanded in the internal work buffer.

tabspec in error
A tab specification is incorrectly formatted, or specified tab stops are not ascending.

tabspec indirection illegal
A *tabspec* read from a file (or standard input) may not contain a *tabspec* referencing another file (or standard input).

0
normal execution

1
for any error

SEE ALSO

csplit(1), tabs(1)
fspec(4) in the *Programmer's Reference Manual.*

BUGS

newform normally only keeps track of physical characters; however, for the **–i** and **–o** options, *newform* will keep track of backspaces in order to line up tabs in the appropriate logical columns.

newform will not prompt the user if a *tabspec* is to be read from the standard input (by use of **–i—** or **–o—**).
If the **–f** option is used, and the last **–o** option specified was **–o—**, and was preceded by either a **–o—** or a **–i—**, the tab specification format line will be incorrect.

NAME

news – print news items

SYNOPSIS

news [**–a**] [**–n**] [**–s**] [*items*]

DESCRIPTION

news is used to keep the user informed of current events. By convention, these events are described by files in the directory **/usr/news**.

When invoked without arguments, *news* prints the contents of all current files in **/usr/news**, most recent first, with each preceded by an appropriate header. *news* stores the "currency" time as the modification date of a file named **.news_time** in the user's home directory (the identity of this directory is determined by the environment variable $HOME); only files more recent than this currency time are considered "current."

–a
option causes *news* to print all items, regardless of currency. In this case, the stored time is not changed.

–n
option causes *news* to report the names of the current items without printing their contents, and without changing the stored time.

–s
option causes *news* to report how many current items exist, without printing their names or contents, and without changing the stored time. It is useful to include such an invocation of *news* in your .profile file, or in the system's **/etc/profile**.

All other arguments are assumed to be specific news items that are to be printed.

If a *delete* is typed during the printing of a news item, printing stops and the next item is started. Another *delete* within one second of the first causes the program to terminate.

FILES

/etc/profile
/usr/news/*
$HOME/.news_time

SEE ALSO

profile(4), environ(5) in the *Programmer's Reference Manual.*

NAME

nice – run a command at low priority

SYNOPSIS

nice [*–increment*] *command* [*arguments*]

DESCRIPTION

nice executes *command* with a lower CPU scheduling priority. If the *increment* argument (in the range 1-19) is given, it is used; if not, an increment of 10 is assumed.

The superuser may run commands with priority higher than normal by using a negative increment, e.g., **—10**.

SEE ALSO

nohup(1)
nice(2) in the *Programmer's Reference Manual*.

DIAGNOSTICS

nice returns the exit status of the subject command.

BUGS

An *increment* larger than 19 is equivalent to 19.

NAME

nl – line numbering filter

SYNOPSIS

nl [**–b***type*] [**–h***type*] [**–f***type*] [**–v**start#] [**–i***incr*] [**–p**] [**–l***num*] [**–s***sep*] [**–w***width*] [**–n***format*] [**–d***delim*] *file*

DESCRIPTION

nl reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

nl views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. 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).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line contents	*Start of*
\:\:\:	header
\:\:	body
\:	footer

Unless optioned otherwise, *nl* assumes the text being read is in a single logical page body.

Command options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

–b*type*

Specifies which logical page body lines are to be numbered. Recognized *types* and their meaning are:

a	number all lines
t	number lines with printable text only
n	no line numbering
p*string*	number only lines that contain the regular expression specified in *string*.

Default *type* for logical page body is **t** (text lines numbered).

–h*type*
Same as **–b***type* except for header. Default *type* for logical page header is **n** (no lines numbered).

–f*type*
Same as **–b***type* except for footer. Default for logical page footer is n (no lines numbered).

–v*start#*
start# is the initial value used to number logical page lines. Default is **1**.

–i*incr*
incr is the increment value used to number logical page lines. Default is **1**.

–p
Do not restart numbering at logical page delimiters.

–l*num*
num is the number of blank lines to be considered as one. For example, **–l2** results in only the second adjacent blank being numbered (if the appropriate **–ha**, **–ba**, and/or **–fa** option is set). Default is **1**.

–s*sep*
sep is the character(s) used in separating the line number and the corresponding text line. Default *sep* is a tab.

–w*width*
width is the number of characters to be used for the line number. Default *width* is **6**.

–n*format*
format is the line numbering format. Recognized values are: **ln**, left justified, leading zeroes suppressed; **rn**, right justified, leading zeroes supressed; **rz**, right justified, leading zeroes kept. Default *format* is rn (right justified).

–d*xx*
The delimiter characters specifying the start of a logical page section may be changed from the default characters (\:) to two user-specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the **–d** and the delimiter characters. To enter a backslash, use two backslashes.

EXAMPLE

The command:

nl –v10 –i10 –d!+ file1

will number *file1* starting at line number 10 with an increment of 10. The logical page delimiters are !+.

SEE ALSO

pr(1)

NAME

nm – print name list of common object file

SYNOPSIS

nm [**–oxhvnefurpVT**] filename ...

DESCRIPTION

The *nm* command displays the symbol table of each common object file, *file*name. *file name* may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, the following information will be printed:

Name
name of the symbol.

Value
value expressed as an offset or an address depending on its storage class.

Class
storage class.

Type
type and derived type. If the symbol is an instance of a structure or of a union then the structure or union tag will be given following the type (e.g., struct-tag). If the symbol is an array, then the array dimensions will be given following the type (e.g., char[n][m]). Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Size
size in bytes, if available. Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Line
source line number at which it is defined, if available. Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.

Section
For storage classes static and external, the object file section containing the symbol (e.g., text, data or bss).

The output of *nm* may be controlled using the following options:

–o
Print the value and size of a symbol in octal instead of decimal.

–x
Print the value and size of a symbol in hexadecimal instead of decimal.

–h
Do not display the output header data.

–v
Sort external symbols by value before they are printed.

–n
Sort external symbols by name before they are printed.

–e
Print only external and static symbols.

–f
Produce full output. Print redundant symbols (.text, .data, .lib, and .bss), normally suppressed.

–u
Print undefined symbols only.

–r
Prepend the name of the object file or archive to each output line.

–p
Produce easily parsable, terse output. Each symbol name is preceded by its value (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **S** (user defined segment symbol), **R** (register symbol), **F** (file symbol), or **C** (common symbol). If the symbol is local (non-external), the type letter is in lowercase.

–V
Print the version of the nm command executing on the standard error output.

–T

By default, *nm* prints the entire name of the symbols listed. Since object files can have symbols names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The **–T** option causes *nm* to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **nm name –e –v** and **nm –ve name** print the static and external symbols in *name*, with external symbols sorted by value.

FILES

TMPDIR/*	temporary files

TMPDIR is usually /usr/tmp but can be redefined by setting the environment variable TMPDIR (see *tempnam()* in *tmpnam*(3S)).

BUGS

When all the symbols are printed, they must be printed in the order they appear in the symbol table in order to preserve scoping information. Therefore, the **–v** and **–n** options should be used only in conjunction with the **–e** option.

SEE ALSO

as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4)

DIAGNOSTICS

"nm: name: cannot open"
if *name* cannot be read.

"nm: name: bad magic"
if *name* is not a common object file.

"nm: name: no symbols"
if the symbols have been stripped from *name*.

NAME

nohup – run a command immune to hangups and quits

SYNOPSIS

nohup *command* [*arguments*]

DESCRIPTION

nohup executes *command* with hangups and quits ignored. If output is not re-directed by the user, both standard output and standard error are sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to **$HOME/nohup.out**.

EXAMPLE

It is frequently desirable to apply *nohup* to pipelines or lists of commands. This can be done only by placing pipelines and command lists in a single file, called a shell procedure. You can then issue:

```
nohup sh file
```

and the *nohup* applies to everything in *file*. If the shell procedure *file* is to be executed often, then the need to type *sh* can be eliminated by giving *file* execute permission. Add an ampersand and the contents of *file* are run in the background with interrupts also ignored (see *sh*(1)):

```
nohup file &
```

An example of what the contents of *file* could be is:

```
sort ofile > nfile
```

SEE ALSO

chmod(1), nice(1), sh(1),
signal(2) in the *Programmer's Reference Manual*.

WARNINGS

In the case of the following command

```
nohup command1; command2
```

nohup applies only to command1. The command

```
nohup (command1; command2)
```

is syntactically incorrect.

NAME

oawk – pattern scanning and processing language

SYNOPSIS

oawk [**–F***c*] [*prog*] [*parameters*] [*files*]

DESCRIPTION

oawk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog*, there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as **–f** *file*. The *prog* string should be enclosed in single quotes (') to protect it from the shell.

parameters, in the form x=... y=... etc., may be passed to *oawk*.

files are read in order; if there are no files, the standard input is read. The file name – means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS; see below). The fields are denoted **$1**, **$2**, . . . , and **$0** refers to the entire line.

A pattern-action statement has the form:

pattern { action }

A missing *action* means print the line; a missing *pattern* always matches. An *action* is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next    # skip remaining patterns on this input line
exit    # skip the rest of the input
```

Statements are terminated by semicolons, newlines, or right braces. An empty *expression-list* stands for the whole line. *expressions* take on string or numeric values as appropriate and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The C operators ++, —, +=, –=, *=, /=, and %= are also available in expressions. *Variables* may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the NULL string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The *print* statement prints its arguments on the standard output (or on a file if *>filename* is present), separated by the current output field separator, and terminated by the output record separator. The *printf* statement formats its *expression-list* as described in *printf*(3S) in the *Programmer's Reference Manual*.

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer; *substr*(*s*, *m*, *n*) returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf*(*fmt*, *expr*, *expr*, ...) formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes, as in *egrep* (see *grep*(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions. A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

expression matchop regular-expression
expression relop expression

where a *relop* is any of the six relational operators in C, and a *matchop* is either ~ (for *contains*) or !~ (for *does not contain*). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields by starting the program with:

BEGIN { FS = *c* }

or by using the **–F***c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default %.6g).

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

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

Add up first column, print sum and average:

```
        { s += $1 }
END     { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

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

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
       { print }
```

command line: oawk –f program n=5 input

SEE ALSO

awk(1), grep(1), sed(1)
lex(1), printf(3S) in the *Programmer's Reference Manual.*

BUGS

Input white space is not preserved on output if fields are involved.
There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the `null` string (" ") to it.

NAME

od – octal dump

SYNOPSIS

od [**–bcdosx**] [*file*] [[**+**]*offset*[**.**][**b**]]

DESCRIPTION

od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing, **–o** is default. The meanings of the format options are:

–b
Interpret bytes in octal.

–c
Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=**\0**, backspace=**\b**, form-feed=**\f**, newline=**\n**, return=**\r**, tab=**\t**; others appear as 3-digit octal numbers.

–d
Interpret words in unsigned decimal.

–o
Interpret words in octal.

–s
Interpret 16-bit words in signed decimal.

–x
Interpret words in hex.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended, 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 **+**.

Dumping continues until end-of-file.

NAME

pack, pcat, unpack – compress and expand files

SYNOPSIS

pack [**–**] [**–f**] *name* ...

pcat *name* ...

unpack *name* ...

DESCRIPTION

pack attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file *name* is replaced by a packed file *name*.**z** with the same access modes, access and modified dates, and owner as those of *name*. The **-f** option will force packing of *name*. This is useful for causing an entire directory to be packed even if some of the files will not benefit. If *pack* is successful, *name* will be removed. Packed files can be restored to their original form using *unpack* or *pcat*.

pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the – argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of – in place of *name* will cause the internal flag to be set and reset.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each **.z** file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Load modules, 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.

pack returns a value that is the number of files that it failed to compress.

No packing will occur if:

the file appears to be already packed;
the file name has more than 12 characters;
the file has links;
the file is a directory;
the file cannot be opened;
no disk storage blocks will be saved by packing;

a file called *name*.z already exists;
the .z file cannot be created;
an I/O error occurred during processing.

The last segment of the file name must contain no more than 12 characters to allow space for the appended .z extension. Directories cannot be compressed.

pcat does for packed files what *cat*(1) does for ordinary files, except that *pcat* cannot be used as a filter. The specified files are unpacked and written to the standard output. Thus, to view a packed file named *name*.z use:

pcat name.z

or just:

pcat name

To make an unpacked copy, say *nnn*, of a packed file named *name*.z (without destroying *name*.z) use the command:

pcat name >nnn

pcat returns the number of files it was unable to unpack. Failure may occur if:

the file name (exclusive of the .z) has more than 12 characters;
the file cannot be opened;
the file does not appear to be the output of *pack*.

unpack expands files created by *pack*. For each file *name* specified in the command, a search is made for a file called *name*.z (or just *name*, if *name* ends in .z). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the .z suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

unpack returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in *pcat*, as well as for the following:

a file with the "unpacked" name already exists;
if the unpacked file cannot be created.

SEE ALSO

cat(1)

NAME

passwd – change login password and password attributes

SYNOPSIS

passwd [*name*]

passwd [**–l** | **–d**] [**–n** *min*] [**–f**] [**–x** *max*] *name*

passwd –s [**–a**]

passwd –s [*name*]

DESCRIPTION

The *passwd* command changes the password or lists password attributes associated with the user's login *name*. Additionally, superusers may use *passwd* to install or change passwords and attributes associated with any login *name*.

When used to change a password, *passwd* prompts ordinary users for their old password, if any. It then prompts for the new password twice. When the old password is entered, *passwd* checks to see if it has "aged" sufficiently. If "aging" is insufficient, *passwd* terminates; see *passwd*(4).

Assuming aging is sufficient, a check is made to ensure that the new password meets construction requirements. When the new password is entered a second time, the two copies of the new password are compared. If the two copies are not identical the cycle of prompting for the new password is repeated for at most two more times.

Passwords must be constructed to meet the following requirements:

Each password must have at least six characters. Only the first eight characters are significant.

Each password must contain at least two alphabetic characters and at least one numeric or special character. In this case, "alphabetic" refers to all upper- or lowercase letters.

Each password must differ from the user's login *name* and any reverse or circular shift of that login *name*. For comparison purposes, an uppercase letter and its corresponding lowercase letter are equivalent.

New passwords must differ from the old by at least three characters. For comparison purposes, an uppercase letter and its corresponding lowercase letter are equivalent.

Superusers (e.g., real and effective uid equal to zero, see *id*(1M) and *su*(1M)) may change any password; hence, *passwd* does not prompt superusers for the old password. Superusers are not forced to comply with password aging and password construction requirements. A superuser can create a NULL password by entering a carriage return in response to the prompt for a new password. (This differs from **passwd –d** because the "password" prompt will still be displayed.)

Any user may use the **–s** option to show password attributes for his or her own login *name*.

The format of the display will be:

name status mm/dd/yy min max

or, if password aging information is not present,

name status

where

name
The login ID of the user.

status
The password status of *name*: "PS" stands for passworded or locked, "LK" stands for locked, and "NP" stands for no password.

mm/dd/yy
The date password was last changed for *name*. (Note that all password aging dates are determined using Greenwich Mean Time and, therefore, may differ by as much as a day in other time zones.)

min
The minimum number of days required between password changes for *name*.

max
The maximum number of days the password is valid for *name*.

Only a superuser can use the following options:

–l
Locks password entry for *name*.

–d
Deletes password for *name*. The login *name* will not be prompted for password.

–n

Set minimum field for *name*. The *min* field contains the minimum number of days between password changes for *name*. If *min* is greater than *max*, the user may not change the password. Always use this option with the **–x** option, unless *max* is set to –1 (aging turned off). In that case, *min* need not be set.

–x

Set maximum field for *name*. The *max* field contains the number of days that the password is valid for *name*. The aging for *name* will be turned off immediately if *max* is set to -1. If it is set to 0, then the user is forced to change the password at the next login session and aging is turned off.

–a

Show password attributes for all entries. Use only with **–s** option; *name* must not be provided.

–f

Force the user to change password at the next login by expiring the password for *name*.

FILES

/etc/passwd, /etc/shadow, /etc/opasswd, /etc/oshadow

SEE ALSO

login(1)
crypt(3C), passwd(4) in the *Programmer's Reference Manual.*
id(1M), passmgmt(1M), pwconv(1M), su(1M), in the *System Administrator's Reference Manual.*

WARNING

If the optional **/etc/shadow** file feature is used, the *passwd*(1) command will use that, instead of the **/etc/passwd** file, to obtain password information. Since the way password aging information is stored in the two files is slightly different, the output from *passwd* options that use this information may also be different.

DIAGNOSTICS

The *passwd* command exits with one of the following values:

0	SUCCESS.
1	Permission denied.
2	Invalid combination of options.
3	Unexpected failure. Password file unchanged.
4	Unexpected failure. Password file(s) missing.
5	Password file(s) busy. Try again later.
6	Invalid argument to option.

NAME

paste – merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste *file1 file2 ...*
paste –d*list file1 file2 ...*
paste –s [–d*list***]** *file1 file2 ...*

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). It is the counterpart of *cat*(1) which concatenates vertically, i.e., one file after the other. In the last form, *paste* replaces the function of an older command with the same name by combining subsequent lines of the input file (serial merging). In all cases, lines are glued together with the *tab* character, or with characters from an optionally specified *list*. Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if – is used in place of a file name.

The meanings of the options are:

–d

Without this option, the newline characters of each but the last file (or last line in the **–s** option) are replaced by a *tab* character. This option allows replacing the *tab* character by one or more alternate characters (see below).

list

One or more characters immediately following **–d** replace the default *tab* as the line concatenation character. The list is used circularly, i.e., when exhausted, it is reused. In parallel merging (i.e., no **–s** option), the lines from the last file are always terminated with a newline character, not from the *list*. The *list* may contain the special escape sequences: **\n** (newline), **\t** (tab), **** (backslash), and **\0** (empty string, not a NULL character). Quoting may be necessary, if characters have special meaning to the shell (e.g., to get one backslash, use *–d* "\\\\").

–s

Merge subsequent lines instead of one from each input file. Use *tab* for concatenation, unless a *list* is specified with **–d** option. Regardless of the *list*, the very last character of the file is forced to be a newline.

–

May be used in place of any file name, to read a line from the standard input. (There is no prompting).

EXAMPLES

ls | paste –d" " –	list directory in one column
ls | paste – – – –	list directory in four columns
paste –s –d"\t\n" file	combine pairs of lines into lines

SEE ALSO

cut(1), grep(1), pr(1)

DIAGNOSTICS

line too long	Output lines are restricted to 511 characters.
too many files	Except for **–s** option, no more than 12 input files may be specified.

NAME

pg – file perusal filter for CRTs

SYNOPSIS

pg [–*number*] [–p *string*] [–cefns] [+*linenumber*] [+/*pattern*/] [*files* ...]

DESCRIPTION

The *pg* command is a filter that allows the examination of *files* one screenful at a time on a CRT. (The file name and/or **NULL** arguments indicate that *pg* should read from the standard input.) Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are described later.

This command is different from previous paginators because it allows you to back up and review something that has already passed. The method for doing this is explained below.

To determine terminal attributes, *pg* scans the *terminfo*(4) data base for the terminal type specified by the environment variable TERM . If TERM is not defined, the terminal type **dumb** is assumed.

The command line options are:

–*number*
An integer specifying the size (in lines) of the window that *pg* is to use instead of the default. (On a terminal containing 24 lines, the default window size is 23).

–p *string*
Causes *pg* to use *string* as the prompt. If the prompt string contains a "%d", the first occurrence of "%d" in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is ":".

–c
Homes the cursor and clears the screen before displaying each page. This option is ignored if **clear_screen** is not defined for this terminal type in the *terminfo*(4) data base.

–e
Causes *pg* **not** to pause at the end of each file.

–f
Normally, *pg* splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results. The **–f** option inhibits *pg* from splitting lines.

–n
Causes an automatic end of command as soon as a command letter is entered. (Normally, commands must be terminated by a **newline** character.)

–s
Causes *pg* to print all messages and prompts in standout mode (usually inverse video).

+*linenumber*
Start up at *linenumber*.

+*/pattern/*
Start up at the first line containing the regular expression pattern.

The responses that may be typed when *pg* pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands that cause further perusal normally take a preceding *address*, an optionally signed number indicating the point from which further text should be displayed. This *address* is interpreted in either pages or lines depending on the command. A signed *address* specifies a point relative to the current page or line, and an unsigned *address* specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are:

(+1) **newline** or **blank**
This causes one page to be displayed. The address is specified in pages.

(+1) **l**
With a relative address this causes *pg* to simulate scrolling the screen, forward or backward, the number of lines specified. With an absolute address this command prints a screenful beginning at the specified line.

(+1) **d** or **^D**
Simulates scrolling half a screen forward or backward.

The following perusal commands take no *address*.

. or **^L**
Typing a single period causes the current page of text to be redisplayed.

$ Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available to search for text patterns in the text. The regular expressions described in *ed*(1) are available. They must always be terminated by a **newline**, even if the *–n* option is specified.

i/*pattern*/
Search forward for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

i^*pattern*^
i?*pattern*?
Search backwards for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately before the current page and continues to the beginning of the current file, without wrap-around. The ^ notation is useful for Adds 100 terminals that will not properly handle the ?.

After searching, *pg* will normally display the line found at the top of the screen. This can be modified by appending **m** or **b** to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix **t** can be used to restore the original situation.

The user of *pg* can modify the environment of perusal with the following commands:

*i***n**
Begin perusing the *i*th next file in the command line. The *i* is an unsigned number, default value is 1.

*i***p**
Begin perusing the *i*th previous file in the command line. *i* is an unsigned number, default is 1.

i **w**
Display another window of text. If *i* is present, set the window size to *i*.

s *filename*
Save the input in the named file. Only the current file being perused is saved. The white space between the **s** and *filename* is optional. This command must always be terminated by a **newline** even if the *–n* option is specified.

h
Help by displaying an abbreviated summary of available commands.

q or **Q**
Quit *pg*.

!*command*
Command is passed to the shell, whose name is taken from the SHELL environment variable. If this is not available, the default shell is used. This command must always be terminated by a **newline** even if the *–n* option is specified.

At any time when output is being sent to the terminal, the user can press the quit key (normally **CTRL**) or the interrupt (**BREAK**) key. This causes *pg* to stop sending output, and display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

If the standard output is not a terminal, then *pg* acts just like *cat*(1), except that a header is printed before each file (if there is more than one).

EXAMPLE

A sample usage of *pg* in reading system news would be

```
news | pg -p (Page %d):
```

NOTES

While waiting for terminal input, *pg* responds to **BREAK DEL**, and ^ by terminating execution. Between prompts, however, these signals interrupt *pgs* current task and place the user in prompt mode. These should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

The *pg* command responds to ^z (CTRL–z) by suspending a job on those systems that provide job control. Upon awakening, the screen is refreshed.

Users of Berkeley's *more* will find that the *z* and *f* commands are available, and that the terminal /, ^, or ? may be omitted from the searching commands.

FILES

/usr/lib/terminfo/?/*	terminal information database
/tmp/pg*	temporary file when input is from a pipe

SEE ALSO

ed(1), grep(1)
terminfo(4) in the *Programmer's Reference Manual* .

BUGS

If terminal tabs are not set every eight positions, undesirable results may occur.

When using *pg* as a filter with another command that changes the terminal I/O options terminal settings may not be restored correctly.

NAME

pr – print files

SYNOPSIS

pr [[**–column**] [**–w***width*] [**–a**]] [**–e***ck*] [**–i***ck*] [**–drtfp**] [**+***page*] [**–n***ck*] [**–o***offset*] [**–l***length*] [**–s***separator*] [**–h***header*] [*file ...*]

pr [[**–m**] [**–w***width*]] [**–e***ck*] [**–i***ck*] [**–drtfp**] [**+***page*] [**–n***ck*] [**–o***offset*] [**–l***length*] [**–s***separator*] [**–h***header*] *file1 file2 ...*

DESCRIPTION

pr is used to format and print the contents of a file. If *file* is –, or if no files are specified, *pr* assumes standard input. *pr* prints the named files on standard output.

By default, the listing is separated into pages, each headed by the page number, the date and time that the file was last modified, and the name of the file. Page length is 66 lines which includes 10 lines of header and trailer output. The header is composed of 2 blank lines, 1 line of text (can be altered with **–h**), and 2 blank lines; the trailer is 5 blank lines.

For single column output, line width may not be set and is unlimited. For multicolumn output, line width may be set and the default is 72 columns. Diagnostic reports (failed options) are reported at the end of standard output associated with a terminal, rather than interspersed in the output. Pages are separated by series of line feeds rather than form feed characters.

By default, columns are of equal width, separated by at least one space; lines which do not fit are truncated. If the **–s** option is used, lines are not truncated and columns are separated by the *separator* character.

Either *–column* or **–m** should be used to produce multi-column output. **–a** should only be used with *–column* and not **–m**.

Command line options are

+page
Begin printing with page numbered *page* (default is 1).

–column
Print *column* columns of output (default is 1). Output appears as if **–e** and **–i** are turned on for multi-column output. May not use with **–m**.

–a
Print multi-column output across the page one line per column. *columns* must be greater than one. If **a** line is too long to fit in a column, it is truncated.

–m
Merge and print all files simultaneously, one per column. The maximum number of files that may be specified is eight. If a line is too long to fit in a column, it is truncated. May not use with *–column.*

–d
Doublespace the output. Blank lines that result from doublespacing are dropped when they occur at the top of a page.

–e*ck*
Expand input tabs to character positions $k+1$, $2*k+1$, $3*k+1$, etc. If k is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If *c* (any non-digit character) is given, it is treated as the input tab character (default for *c* is the tab character).

–i*ck*
In output, replace white space wherever possible by inserting tabs to character positions $k+1$, $2*k+1$, $3*k+1$, etc. If k is 0 or is omitted, default tab settings at every eighth position are assumed. If *c* (any non-digit character) is given, it is treated as the output tab character (default for *c* is the tab character).

–n*ck*
Provide *k*-digit line numbering (default for *k* is 5). The number occupies the first $k+1$ character positions of each column of single column output or each line of **–m** output. If *c* (any non-digit character) is given, it is appended to the line number to separate it from whatever follows (default for *c* is a tab).

–w*width*
Set the width of a line to *width* character positions (default is 72). This is effective only for multi-column output (*-column* and **–m**). There is no line limit for single column output.

–o*offset*
Offset each line by *offset* character positions (default is 0). The number of character positions per line is the sum of the width and offset.

–l*length*
Set the length of a page to *length* lines (default is 66). –l0 is reset to –l66. When the value of *length* is 10 or less, **–t** appears to be in effect since headers and trailers are suppressed. By default, output contains 5 lines of header and 5 lines of trailer leaving 56 lines for user-supplied text. When **–l***length* is used and *length* exceeds 10, then *length*-10 lines are left per page for user supplied text. When *length* is 10 or less, header and trailer output is omitted to make room for user supplied text.

–h *header*
Use *header* as the text line of the header to be printed instead of the file name. **–h** is ignored when **–t** is specified or **–l***length* is specified and the value of *length* is 10 or less. **(–h is the only** *pr* option requiring space between the option and argument.)

–p
Pause before beginning each page if the output is directed to a terminal (*pr* will ring the bell at the terminal and wait for a carriage return).

–f
Use single form-feed character for new pages (default is to use a sequence of linefeeds). Pause before beginning the first page if the standard output is associated with a terminal.

–r
Print no diagnostic reports on files that will not open.

–t
Print neither the five-line identifying header nor the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page. Use of **–t** overrides the **–h** option.

–s*separator*
Separate columns by the single character *separator* instead of by the appropriate number of spaces (default for *separator* is a tab). Prevents truncation of lines on multicolumn output unless **–w** is specified.

EXAMPLES

Print *file1* and *file2* as a doublespaced, threecolumn listing headed by "file list":

pr –3dh "file list" file1 file2

Copy *file1* to *file2*, expanding tabs to columns 10, 19, 28, 37, . . . :

pr –e9 –t <file1 >file2

Print *file1* and *file2* simultaneously in a twocolumn listing with no header or trailer where both columns have line numbers:

pr –t –n file1 | pr –t –m –n file2 -

FILES

/dev/tty* If standard output is directed to one of the special files **/dev/tty***, then 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), pg(1)

NAME

prof – display profile data

SYNOPSIS

prof [**–tcan**] [**–ox**] [**–g**] [**–z**] [**–h**] [**–s**] [**–m***mdata*] [*prog*]

DESCRIPTION

The *prof* command interprets a profile file produced by the *monitor*(3C) function. The symbol table in the object file *prog* (**a.out** by default) is read and correlated with a profile file (**mon.out** by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options **t**, **c**, **a**, and **n** determine the type of sorting of the output lines:

–t
Sort by decreasing percentage of total time (default).

–c
Sort by decreasing number of calls.

–a
Sort by increasing symbol address.

–n
Sort lexically by symbol name.

The mutually exclusive options **o** and **x** specify the printing of the address of each symbol monitored:

–o
Print each symbol address (in octal) along with the symbol name.

–x
Print each symbol address (in hexadecimal) along with the symbol name.

The following options may be used in any combination:

–g
Include non-global symbols (static functions).

–z
Include all symbols in the profile range (see *monitor*(3C)), even if associated with zero number of calls and zero time.

–h
Suppress the heading normally printed on the report. (This is useful if the report is to be processed further.)

–s
Print a summary of several of the monitoring parameters and statistics on the standard error output.

–m mdata
Use file *mdata* instead of **mon.out** as the input profile file.

A program creates a profile file if it has been loaded with the **–p** option of *cc*(1). This option to the *cc* command arranges for calls to *monitor*(3C) at the beginning and end of execution. It is the call to *monitor* at the end of execution that causes a profile file to be written. The number of calls to a function is tallied if the **–p** option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environment variable PROFDIR. If PROFDIR does not exist, "mon.out" is produced in the directory that is current when the program terminates. If PROFDIR = string, "string/pid.progname" is produced, where *progname* consists of *argv*[0] with any path prefix removed, and *pid* is the program's process id. If PROFDIR is the **NULL** string, no profiling output is produced.

A single function may be split into subfunctions for profiling by means of the MARK macro (see *prof*(5)).

FILES

mon.out	for profile
a.out	for namelist

SEE ALSO

cc(1), exit(2), profil(2), monitor(3C), prof(5)

WARNING

The times reported in successive identical runs may show variances of 20% or more, because of varying cache-hit ratios due to sharing of the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data. In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements.

Call counts are always recorded precisely.

The times for static functions are attributed to the preceding external text symbol if the -g option is not used. However, the call counts for the preceding function are still correct, i.e., the static function call counts are not added in with the call counts of the external function.

CAVEATS

Only programs that call *exit*(2) or return from *main* will cause a profile file to be produced, unless a final call to monitor is explicitly coded.

The use of the **–p** option to *cc*(1) to invoke profiling imposes a limit of 600 functions that may have call counters established during program execution. For more counters you must call *monitor*(3C) directly. If this limit is exceeded, other data will be overwritten and the **mon.out** file will be corrupted. The number of call counters used will be reported automatically by the *prof* command whenever the number exceeds 5/6 of the maximum.

NAME

prs – print an SCCS file

SYNOPSIS

prs [**–d**[*dataspec*]] [**–r**[*SID*]] [**–e**] [**–l**] [**–c**[*date-time*]] [**–a**] *files*

DESCRIPTION

prs prints, on the standard output, parts or all of an SCCS file (see *sccsfile*(4)) in a user-supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**), and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of *keyletter* arguments, and file names.

All the described *keyletter* arguments apply independently to each named file:

–d*[dataspec]*
Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see **DATA KEYWORDS**) interspersed with optional user supplied text.

–r*[SID]*
Used to specify the *SCCS* ID*entification* (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

–e
Requests information for all deltas created *earlier* than and including the delta designated via the **–r** keyletter or the date given by the –c option.

–l
Requests information for all deltas created *later* than and including the delta designated via the **–r** keyletter or the date given by the –c option.

c date-time The cutoff date-time **–c**[cutoff] is in the form:

YY[MM[DD[HH[MM[SS]]]]]

–c*[date-time]*

Units omitted from the date-time default to their maximum possible values; that is, **–c7502** is equivalent to **-c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date in the form: **"–c77/2/2 9:22:25"**.

–a

Requests printing of information for both removed, i.e., delta type = *R*, (see *rmdel*(1)) and existing, i.e., delta type = *D*, deltas. If the –a keyletter is not specified, information for existing deltas only is provided.

DATA KEYWORDS

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(4)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User-supplied text is any text other than recognized data keywords.

A tab is specified by **\t** and carriage return/newline is specified by **\n**. The default data keywords are:

":Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:"

Table 1. SCCS Files Data Keywords

Keyword	Data Item	File Section	Value	Format
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D*~or~*R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created		nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS:~:DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS:~:DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS:~:DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes*~or~*no*	S

Table 1. SCCS Files Data Keywords (cont'd)

Keyword	Data Item	File Section	Value	Format
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes*˜or˜*no*	S
:KV:	Keyword validation string	"	text	S
:BF:	Branch flag	"	*yes*˜or˜*no*	S
:J:	Joint edit flag	"	*yes*˜or˜*no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User-defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes*˜or˜*no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y:˜:M:˜:I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file pathname	N/A	text	S

* :Dt:˜=˜:DT:˜:I:˜:D:˜:T:˜:P:˜:DS:˜:DP:

EXAMPLES

prs –d"Users and/or user IDs for :F: are:\n:UN:" s.file

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

prs –d"Newest delta for pgm :M:: :I: Created :D: By :P:" –r s.file

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1
By cas
```

As a *special case:*

prs s.file

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only keyletter argument allowed to be used with the *special case* is the **–a** keyletter.

FILES

/tmp/pr?????

SEE ALSO

admin(1), delta(1), get(1), sccsfile(4)
help(1) in the *User's Reference Manual.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ps – report process status

SYNOPSIS

ps [*options*]

DESCRIPTION

The *ps* command prints certain information about active processes. Without *options*, information is printed about processes associated with the controlling terminal. The output consists of a short listing containing only the process ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the selection of *options*.

The *options* accept names or lists as arguments. Arguments can be either separated from one another by commas or enclosed in double quotes and separated from one another by commas or spaces. Values for *proclist* and *grplist* must be numeric.

The *options* are given in descending order according to volume and range of information provided:

–e
Report on every process now running.

–d
Report on all processes except process group leaders.

–a
Report on all processes most frequently requested: all those except process group leaders and processes not associated with a terminal.

–f
Generate a full listing. (See below for significance of columns in a full listing.)

–l
Generate a long listing. (See below.)

–n *name*
Valid only for users with a real user id of *root* or a real group id of *sys*. Takes argument signifying an alternate system *name* in place of **/unix**.

–t *termlist*
List only process data associated with the terminal given in *termlist*. Terminal identifiers may be specified in one of two forms: the device's file name (e.g., **tty04**) or, if the device's file name starts with **tty**, just the digit identifier (e.g., **04**).

–p *proclist*
List only process data whose process ID numbers are given in *proclist*.

–u *uidlist*
List only process data whose user ID number or login name is given in *uidlist*. In the listing, the numerical user ID will be printed unless you give the **–f** option, which prints the login name.

–g *grplist*
List only process data whose process group leader's ID numbers appear in *grplist*. (A group leader is a process whose process ID number is identical to its process group ID number. A login shell is a common example of a process group leader.)

Under the **–f** option, *ps* tries to determine the command name and arguments given when the process was created by examining the user block. Failing this, the command name is printed, as it would have appeared without the **–f** option, in square brackets.

The column headings and the meaning of the columns in a *ps* listing are given below; the letters **f** and **l** indicate the option (full or long, respectively) that causes the corresponding heading to appear; **all** means that the heading always appears. Note that these two options determine only what information is provided for a process; they do not determine which processes will be listed.

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

Motorola 88K Computer

00	Process has terminated: process table entry 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 **ptrace**(2).
08	Process cannot wake up by a signal.
10	Process currently in core.
20	Process cannot be swapped.
40	Set when signal goes remote.
80	Process in stream poll.

S (l) The state of the process:

[0,1..N-1]
Process is currently running on system processor whose number is displayed.

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.

A Awaiting: process is waiting for an atomic event to complete.

UID (f,l) The user ID number of the process owner (the login name is printed under the **–f** option).

PID (all) The process ID of the process (this datum is necessary to kill a process).

PPID (f,l) The process ID of the parent process.

C (f,l) Processor utilization for scheduling.

PRI (l) The priority of the process (higher numbers mean lower priority).

NI (l) Nice value, used in priority computation.

ADDR (l) The memory address of the process.

SZ (l) The size (in pages or clicks) of the swappable process's image in main memory.

WCHAN (l) The address of an event for which the process is sleeping or in SXBRK state (if blank, the process is running). For awaiting processes, it is a pointer to the waitchannel structure.

STIME	(f)	The starting time of the process, given in hours, minutes, and seconds. (A process begun more than twenty-four hours before the **ps** inquiry is executed is given in months and days.)
TTY	(all)	The controlling terminal for the process (the message, **?**, is printed when there is no controlling terminal).
TIME	(all)	The cumulative execution time for the process.
COMMAND	(all)	The command name (the full command name and its arguments are printed under the **–f** option).

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked **<defunct>**.

FILES

/dev	
/dev/sxt/*	
/dev/tty*	
/dev/xt/*	terminal (''tty'') names searcher files
/dev/kmem	kernel virtual memory
/dev/swap	the default swap device
/dev/mem	memory
/etc/passwd	UID information supplier
/etc/ps_data	internal data structure
/unix	system namelist

SEE ALSO

kill(1), nice(1)
getty(1M) in the *System Administrator's Reference Manual.*

WARNING

Things can change while *ps* is running; the snap-shot it gives is only true for a split-second, and it may not be accurate by the time you see it. Some data printed for defunct processes is irrelevant.

If no *termlist*, *proclist*, *uidlist*, or *grplist* is specified, *ps* checks **stdin**, **stdout**, and **stderr** in that order, looking for the controlling terminal and will attempt to report on processes associated with the controlling terminal. In this situation, if **stdin**, **stdout**, and **stderr** are all redirected, *ps* will not find a controlling terminal, so there will be no report.

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 a process' user area, **ps** may not be able to seek to that address before the process exits and the address becomes invalid.

ps **–ef** may not report the actual start of a TTY login session, but rather an earlier time, when a getty was last respawned on the tty line.

If the user specifies the **–n** flag, the real and effective UID/GID will be set to the real UID/GID of the user invoking **ps**.

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

pwd prints the path name of the working (current) directory.

SEE ALSO

cd(1)

DIAGNOSTICS

"Cannot open .." and "Read error in .." indicate possible file system trouble and should be referred to your local system administrator.

NAME

real – echo the real device in the **permissions** file for a given alias

SYNOPSIS

real [**-r**] [*name*]

DESCRIPTION

real has an optional argument *name*. This argument is used to search the **permissions** file and determine the real device that would be accessed. The file is searched, and when *name* matches either the *slice* or *alias* field of a **permissions** file entry, the pathname of the real device is echoed. This device pathname is used by the disk access routines *chk*, *mnt* and *umnt*.

The default value for *name* is **floppy**.

The optional **-r** argument indicates that the real device desired is the raw device. This device pathname is used by the routines *dcpy* and *fs*.

FILES

/etc/real
/etc/filesys **permissions** file

SEE ALSO

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

NAME

regcmp – regular expression compile

SYNOPSIS

regcmp [**–**] *files*

DESCRIPTION

The *regcmp* command performs a function similar to *regcmp*(3X) and, in most cases, precludes the need for calling *regcmp*(3X) from C programs. This saves on both execution time and program size. The command *regcmp* compiles the regular expressions in *file* and places the output in *file*.**i**. If the **–** option is used, the output will be placed in *file*.c. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotes. The output of *regcmp* is C source code. Compiled regular expressions are represented as **extern char** vectors. *file*.**i** files may thus be *included* in C programs, or *file*.**c** files may be compiled and later loaded. In the C program which uses the *regcmp* output, *regex*(*abc*,*line*) will apply the regular expression named *abc* to *line*. Diagnostics are self-explanatory.

EXAMPLES

```
name    "([A–Za–z][A–Za–z0–9_]*)$0"

telno   "\({0,1}([2–9][01][1–9])$0\){0,1} *"
        "([2–9][0–9]{2})$1[ –]{0,1}"
        "([0–9]{4})$2"
```

In the C program that uses the *regcmp* output,

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named *telno* to *line*.

SEE ALSO

regcmp(3X)

NAME

rlogin – remote login.

SYNOPSIS

rlogin *host* [**–e***x*] [**–l** *username*] [**–8**]

DESCRIPTION

rlogin connects your terminal on the current local host system lhost to the remote host system rhost.

Each host has a file **/etc/hosts.equiv** that contains a list of rhosts with which it shares account names. (The host names must be the standard names as described in *remsh*(1).) When you *rlogin* as the same user on an equivalent host, you may not need to give a password (depending on the restrictions imposed by *login*(1) on the remote host). Each user may also have a private equivalence list in a file **.rhosts** in his login directory. Each line in this file should contain a rhost and a username separated by a space, giving additional cases where logins without passwords may be permitted. If the originating user is not equivalent to the remote user, then a login and password will be prompted for on the remote machine as in *login*(1). To avoid some security problems, the **.rhosts** file must be owned by either the remote user or root.

Your remote terminal type is the same as your local terminal type (as given in your environment TERM variable). All echoing takes place at the remote site, so that (except for delays) the *rlogin* is transparent. Flow control via ^S and ^Q and flushing of input and output on interrupts are handled properly.

A line of the form ~ . (where ~ is the escape character) disconnects from the remote host. A different escape character may be specified by the **–e***x* option. There is no space separating this option flag and the argument character *x*.

While connected, an input string of the form ~ ! introduces an arbitrary command to be executed locally (à la *cu*(1)). ~ ! alone spawns a local shell. ~ ! must be preceded by a <CR> (i.e., must be the first two characters of a new input line).

Using the **-8** option configures the connection for eight-bit data transfers (seven-bit data is the default).

SEE ALSO

remsh(1)

FILES

/etc/hosts.equiv
$HOME/.rhosts

BUGS

More terminal characteristics should be propagated.

When connecting to SYSTEM V/88 systems running NSE, users may be required to enter their password (if they have one) on the remote system. This is due to the operation of login on these systems.

NAME

rm, rmdir – remove files or directories

SYNOPSIS

rm [**–f**] [**–i**] *file* ...

rm –r [**–f**] [**–i**] *dirname* ... [*file* ...]

rmdir [**–p**] [**–s**] *dirname* ...

DESCRIPTION

rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. If a file has no write permission and the standard input is a terminal, the full set of permissions (in octal) for the file are printed followed by a question mark. This is a prompt for confirmation. If the answer begins with **y** (for yes), the file is deleted, otherwise the file remains.

When the target is a symbolic link, *rm* removes the symbolic link itself, not the file that the link points to.

Note that if the standard input is not a terminal, the command will operate as if the **–f** option is in effect.

Three options apply to *rm*:

–f
: This option causes the removal of all files (whether write-protected or not) in a directory without prompting the user. In a write-protected directory, however, files are never removed (whatever their permissions are), but no messages are displayed. If the removal of a write-protected directory is attempted, this option will not suppress an error message.

–r
: This option causes the recursive removal of any directories and subdirectories in the argument list. The directory will be emptied of files and removed. Note that the user is normally prompted for removal of any write-protected files which the directory contains. The write-protected files are removed without prompting, however, if the **–f** option is used, or if the standard input is not a terminal and the **–i** option is not used.

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

With this option, confirmation of removal of any write-protected file occurs interactively. It overrides the **–f** option and remains in effect even if the standard input is not a terminal.

Two options apply to *rmdir*:

–p

This option allows users to remove the directory *dirname* and its parent directories which become empty. A message is printed on standard output as to whether the whole path is removed or part of the path remains for some reason.

–s

This option is used to suppress the message printed on standard error when **–p** is in effect.

DIAGNOSTICS

All messages are generally self-explanatory.

It is forbidden to remove the files "." and ".." in order to avoid the consequences of inadvertently doing something like the following:

```
rm –r .*
```

Both *rm* and *rmdir* return exit codes of 0 if all the specified directories are removed successfully. Otherwise, they return a non-zero exit code.

SEE ALSO

unlink(2), rmdir(2) in the *Programmer's Reference Manual*.

NAME

rmdel – remove a delta from an SCCS file

SYNOPSIS

rmdel –r*SID files*

DESCRIPTION

rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the specified must *not* be that of a version being edited for the purpose of making a delta (i. e., if a **p-file** (see *get*(1)) exists for the named SCCS file, the specified must *not* appear in any entry of the **p-file**).

The **–r** option is used for specifying the *SID* level of the delta to be removed.

If a directory is named, *rmdel* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

x.file (see *delta*(1))
z.file (see *delta*(1))

SEE ALSO

delta(1), get(1), prs(1), sccsfile(4)
help(1) in the *User's Reference Manual*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sact – print current SCCS file editing activity

SYNOPSIS

sact *files*

DESCRIPTION

sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(1) with the **–e** option has been previously executed without a subsequent execution of *delta*(1). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1
: specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

Field 2
: specifies the SID for the new delta to be created.

Field 3
: contains the logname of the user who will make the delta (i.e., executed a *get* for editing).

Field 4
: contains the date that **get –e** was executed.

Field 5
: contains the time that **get –e** was executed.

SEE ALSO

delta(1), get(1), unget(1)

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sar – system activity reporter

SYNOPSIS

sar [**–uUbdycwaqvmMprDSAPK**] [**–o** *file*] *t* [*n*]

sar [**–uUbdycwaqvmMprDSAPK**] [**–s** *time*] [**–e** *time*] [**–i** *sec*] [**–f** *file*]

DESCRIPTION

sar, in the first instance, samples cumulative activity counters in the operating system at *n* intervals of *t* seconds, where *t* should be 5 or greater. (If the sampling interval is less than 5, the activity of *sar* itself may affect the sample.) If the **–o** option is specified, it saves the samples in *file* in binary format. The default value of *n* is 1. In the second instance, with no sampling interval specified, *sar* extracts data from a previously recorded *file*, either the one specified by the **–f** option or, by default, the standard system activity daily data file /usr/adm/sa/sa*dd* for the current day *dd*. The starting and ending times of the report can be bounded via the **–s** and **–e** *time* arguments of the form *hh*[:*mm*[:*ss*]]. The **–i** option selects records at *sec* second intervals. Otherwise, all intervals found in the data file are reported.

In either case, subsets of data to be printed are specified by option:

–u
Report CPU utilization (the default):
%usr, %sys, %wio, %idle – portion of time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle. When used with **–D**, %sys is split into percent of time servicing requests from remote machines (%sys remote) and all other system time (%sys local).

–U Report
CPU utilization for each active system processor and a total for all processors:
CPU id#, %usr, %sys, %wio, %idle – the processor id number, portion of time it was running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle. When used with **–D**, %sys is split into percent of time servicing requests from remote machines (%sys remote) and all other system time (%sys local). Lastly, the average for all system processors is printed. This line will be identical to that reported by the **–u** option.

–b

Report buffer activity:
bread/s, bwrit/s – transfers per second of data between system buffers and disk or other block devices;
lread/s, lwrit/s – accesses of system buffers;
%rcache, %wcache – cache hit ratios, i. e., (1–bread/lread) as a percentage;
pread/s, pwrit/s – transfers via raw (physical) device mechanism. When used with **–D**, buffer caching is reported for locally-mounted remote resources.

–d

Report activity for each block device, e. g., disk or tape drive, with the exception of XDC disks and tape drives. When data is displayed, the device specification *dsk-* is generally used to represent a disk drive. The device specification used to represent a tape drive is machine dependent. The activity data reported is:

%busy, avque – portion of time device was busy servicing a transfer request, average number of requests outstanding during that time;
r+w/s, blks/s – number of data transfers from or to device, number of bytes transferred in 512-byte units;
avwait, avserv – average time in ms. that transfer requests wait idly on queue, and average time to be serviced (which for disks includes seek, rotational latency and data transfer times).

–y

Report TTY device activity:
rawch/s, canch/s, outch/s – input character rate, input character rate processed by canon, output character rate;
rcvin/s, xmtin/s, mdmin/s – receive, transmit and modem interrupt rates.

–c

Report system calls:
scall/s – system calls of all types;
sread/s, swrit/s, fork/s, exec/s – specific system calls;
rchar/s, wchar/s – characters transferred by read and write system calls. When used with **–D**, the system calls are split into incoming, outgoing, and strictly local calls.

–w
Report system swapping and switching activity:
swpin/s, swpot/s, bswin/s, bswot/s – number of transfers and number of 512-byte units transferred for swapins and swapouts (including initial loading of some programs);
pswch/s – process switches.

–a
Report use of file access system routines:
iget/s, namei/s, dirblk/s.

–q
Report average queue length while occupied, and % of time occupied:
runq-sz, %runocc – run queue of processes in memory and runnable;
swpq-sz, %swpocc – swap queue of processes swapped out but ready to run.

–v
Report status of process, i-node, file tables:
text-sz, proc-sz, inod-sz, file-sz, lock-sz – entries/size for each table, evaluated once at sampling point;
ov – overflows that occur between sampling points for each table.

–m
Report message and semaphore activities:
msg/s, sema/s – primitives per second.

–M
Report multiprocessor information for each active processor:
CPU id# - processor id number;
scall/s - system calls of all types for this processor;
pgflt/s - page faults for this processor;
resch/s - process rescheduling for this processor;
pswch/s - process switch for this processor;
traps/s - traps (other than system calls) for this processor;
intrp/s - interrupts for this processor.

–p
Report paging activities:
vflt/s – address translation page faults (valid page not in memory);
pflt/s – page faults from protection errors (illegal access to page) or "copy-on-writes";
pgfil/s – vflt/s satisfied by page-in from file system;
rclm/s – valid pages reclaimed for free list.

–r

Report unused memory pages and disk blocks:
freemem – average pages available to user processes;
freeswap – disk blocks available for process swapping.

–D

Report Remote File Sharing activity:
When used in combination with **–u**, **–U**, **–b** or **–c**, it causes **sar** to produce the remote file sharing version of the corresponding report. **–Du** is assumed when only **–D** is specified.

–S

Report server and request queue status:
Average number of Remote File Sharing servers on the system (serv/lo-hi), % of time receive descriptors are on the request queue (request %busy), average number of receive descriptors waiting for service when queue is occupied (request avg lgth), % of time there are idle servers (server %avail), average number of idle servers when idle ones exist (server avg avail).

–P

Report overall parallelism of all active system processors:
This option reports how much parallelism of execution is being achieved by all active system processors. It reports the percent of time that: zero, one, two, three, and four processors were found busy. Do not interpret these numbers as CPU id numbers. These percentages represent the total number of system processors found simultaneously busy doing useful work. For example, if two processors are found busy 100% of the time, this implies that any combination of two system processors were always found busy. It does NOT mean that processor id# two was busy. If zero processors are found busy 100% of the time, your system is idle.

–K

Report the contention of all active processors for the kernel semaphore (lock):
The multiprocessor kernel is a peer-processor semaphored architecture. All processors are equal, each capable of running both system and user code. The only restriction prohibits two or more processors from running in the kernel at the same time. To prevent this, a semaphore has been placed at those kernel entry points that represent the beginning of system execution. This option reports the amount of contention each processor encounters when it attempts to enter the kernel to start execution of system code.
CPU id# number - the processor's id number;
enters/s - number of times this processor enter the kernel;
%waiting - percentage of time this processor had to wait before being allowed to enter.

–A

Report all data. Equivalent to **–uUdqbwcayvmMprSDPK**.

EXAMPLES

To see today's CPU activity so far:

```
sar
```

To watch CPU activity evolve for 10 minutes and save data:

```
sar –o temp 60 10
```

To later review disk and tape activity from that period:

```
sar –d –f temp
```

FILES

/usr/adm/sa/sa*dd* daily data file, where *dd* are digits representing the day of the month.

SEE ALSO

sag(1G), sar(1M)

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff **–r***SID*1 **–r***SID*2 **[–p]** **[–s***n***]** *files*

DESCRIPTION

sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

–r*SID?*
SID1 and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to *bdiff*(1) in the order given.

–p
pipe output for each file through *pr*(1).

–s*n*
n is the file segment size that *bdiff* will pass to *diff*(1). This is useful when *diff* fails due to a high system load.

FILES

/tmp/get????? Temporary files

SEE ALSO

get(1)
bdiff(1), help(1), pr(1) in the *User's Reference Manual.*

DIAGNOSTICS

"*file*: No differences" If the two versions are the same.
Use *help*(1) for explanations.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [**–u**] [**–w**] [**–W**] [*objfil* [*corfil* [*directory-list*]]]

DESCRIPTION

The *sdb* command calls a symbolic debugger that can be used with C programs compiled with the GreenHills 88000 compiler. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

objfil is an executable program file that has been compiled with the **–g** (debug) option. If it has not been compiled with the **–g** option, the symbolic capabilities of *sdb* will be limited, but the file can still be examined and the program debugged. The default for *objfil* is **a.out**. *corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core**. The core file need not be present. A **–** in place of *corfil* will force *sdb* to ignore any core image file. The colon separated list of directories (*directory-list*) is used to locate the source files used to build *objfil*.

It is useful to know that at any time there is a *current line* and *current file*. If *corfil* exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in *main()*. The current line and file may be changed with the source file examination commands.

By default, warnings are provided if the source files used in producing *objfil* cannot be found, or are newer than *objfil*. This checking feature and the accompanying warnings may be disabled by the use of the **–W** flag.

Names of variables are written just as they are in C. *sdb* does not truncate names. Variables local to a procedure may be accessed using the form *procedure:variable*. If no procedure name is given, the procedure containing the current line is used by default.

Current compilers prepend an underscore character to all variable names, *sdb* allows the user to optionally supply the leading underscore. If the **–u** flag is given then *sdb* will always require the leading underscore on variable names and will always print names in the same fashion.

It is also possible to refer to structure members as *variable.member*, pointers to structure members as *variable->member* and array elements as *variable*[*number*]. Pointers may be dereferenced by using the form *pointer*[0]. Combinations of these forms may also be used.

F77 common variables may be referenced by using the name of the common block instead of the structure name. Blank common variables may be named by the form *.variable*. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure referenced by *sdb*. An unqualified structure variable may also be used with various commands.

Generally, *sdb* will interpret a structure as a set of variables. Thus, *sdb* will display the values of all the elements of a structure when it is requested to display a structure. An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value *sdb* displays, not the addresses of individual elements.

Elements of a multidimensional array may be referenced as *variable* [*number*][*number*]..., or as *variable* [*number,number,...*]. In place of *number*, the form *number;number* may be used to indicate a range of values, * may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. As with structures, *sdb* displays all the values of an array or of the section of an array if trailing subscripts are omitted. It displays only the address of the array itself or of the section specified by the user if subscripts are omitted. A multidimensional parameter in an F77 program cannot be displayed as an array, but it is actually a pointer, whose value is the location of the array. The array itself can be accessed symbolically from the calling function.

A particular instance of a variable on the stack may be referenced by using the form *procedure:variable,number*. All the variations mentioned in naming variables may be used. *number* is the occurrence of the specified procedure on the stack, counting the top, or most current, as the first. If no procedure is specified, the procedure currently executing is used by default.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as *file-name:number* or *procedure:number*. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under *sdb*, all addresses refer to the executing program; otherwise they refer to *objfil* or *corfil*. An initial argument of –w permits overwriting locations in *objfil*.

Addresses

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1*, *e1*, *f1*) and (*b2*, *e2*, *f2*) and the *file address* corresponding to a written *address* is calculated as follows:

$b1 <= address < e1$

$file\ address = address + f1 - b1$

otherwise

$b2 <= address < e2$

$file\ address = address + f2 - b2,$

Otherwise, the requested *address* is not legal. In some cases (e.g., for programs with separated I and D space), the two segments for a file may overlap.

The initial setting of both mappings is suitable for normal **a.out** and core files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size, and *f1* is set to 0; in this way the whole file can be examined with no address translation.

In order for *sdb* to be used on large files, all appropriate values are kept as signed 32-bit integers.

Commands

The commands for examining data in the program are:

t

Print a stack trace of the terminated or halted program.

T

Print the top line of the stack trace.

variable/*clm*

Print the value of *variable* according to length *l* and format *m*. A numeric count *c* indicates that a region of memory, beginning at the address implied by *variable*, is to be displayed. The length specifiers are:

b	one byte
h	two bytes (half word)
l	four bytes (long word)

Legal values for *m* are:

c	character
d	decimal
u	decimal, unsigned
o	octal
x	hexadecimal
f	32-bit single precision floating point
g	64-bit double precision floating point
s	Assume *variable* is a string pointer and print characters starting at the address pointed to by the variable.
a	Print characters starting at the variable's address. This format may not be used with register variables.
p	pointer to procedure
i	disassemble machine-language instruction with addresses printed numerically and symbolically.
I	disassemble machine-language instruction with addresses just printed numerically.

Length specifiers are only effective with the **c**, **d**, **u**, **o** and **x** formats. Any of the specifiers, *c*, *l*, and *m*, may be omitted. If all are omitted, *sdb* chooses a length and a format suitable for the variable's type as declared in the program. If *m* is specified, this format is used for displaying the variable. A length specifier determines the output length of the value to be displayed, sometimes resulting in truncation. A count specifier *c* tells *sdb* to display that many units of memory, beginning at the address of *variable*. The number of bytes in one such unit of memory is determined by the length specifier *l*, or if no length is given, by the size associated with the *variable*. If a count specifier is used for the **s** or **a** command, then that many characters are printed. Otherwise, successive characters are printed until either a NULL byte is reached or 128 characters are printed. The last variable may be redisplayed with the command **./**.

The *sh*(1) metacharacters * and ? may be used within procedure and variable names, providing a limited form of pattern matching. If no procedure name is given, variables local to the current procedure and global variables are matched; if a procedure name is specified then only variables local to that procedure are matched. To match only global variables, the form *:pattern* is used.

linenumber?*lm*
variable:?*lm*
Print the value at the address from **a.out** or I space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is 'i'.

variable=*lm*
linenumber=*lm*
number=*lm*
Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

variable!*value*
Set *variable* to the given *value*. The value may be a number, a character constant or a variable. The value must be well defined; expressions which produce more than one value, such as structures, are not allowed. Character constants are denoted *'character*.

Numbers are viewed as integers unless a decimal point or exponent is used. In this case, they are treated as having the type double. Registers are viewed as integers. The *variable* may be an expression that indicates more than one variable, such as an array or structure name. If the address of a variable is given, it is regarded as the address of a variable of type *int*. C conventions are used in any type conversions necessary to perform the indicated assignment.

x
Print the machine registers and the current machine-language instruction.

X
Print the current machine-language instruction.

The commands for examining source files are:

e *procedure*
e *file-name*
e *directory/*
e *directory file-name*
The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure name and file name are reported.

/regular expression/
Search forward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing / may be deleted.

?regular expression?
Search backward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing ? may be deleted.

p
Print the current line.

z
Print the current line followed by the next 9 lines. Set the current line to the last line printed.

w
Window. Print the 10 lines around the current line.

number
Set the current line to the given line number. Print the new current line.

count **+**
Advance the current line by *count* lines. Print the new current line.

count **−**
Retreat the current line by *count* lines. Print the new current line.

The commands for controlling the execution of the source program are:

count **r** *args*
count **R**

Run the program with the given arguments. The *r* command with no arguments reuses the previous arguments to the program while the *R* command runs the program with no arguments. An argument beginning with < or > causes redirection for the standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber **c** *count*
linenumber **C** *count*

Continue after a breakpoint or interrupt. If *count* is given, the program will stop when *count* breakpoints have been encountered. The signal which caused the program to stop is reactivated with the **C** command and ignored with the **c** command. If a line number is specified then a temporary breakpoint is placed at the line and execution is continued. The breakpoint is deleted when the command finishes.

linenumber **g** *count*

Continue after a breakpoint with execution resumed at the given line. If *count* is given, it specifies the number of breakpoints to be ignored.

s *count*
S *count*

Single step the program through *count* lines. If no count is given then the program is run for one line. **S** is equivalent to **s** except it steps through procedure calls.

i
I Single step by one machine-language instruction. The signal which caused the program to stop is reactivated with the **I** command and ignored with the **i** command.

variable$**m** *count*
address:**m** *count*

Single step (as with **s**) until the specified location is modified with a new value. If *count* is omitted, it is effectively infinity. *Variable* must be accessible from the current procedure. Since this command is done by software, it can be very slow.

level **v**

Toggle verbose mode, for use when single stepping with **S**, **s** or **m**. If *level* is omitted, then just the current source file and/or subroutine name is printed when either changes. If *level* is 1 or greater, each C source line is printed before it is executed; if *level* is 2 or greater, each assembler statement is also printed. A **v** turns verbose mode off if it is on for any level.

k

Kill the program being debugged.

procedure(arg1,arg2,...)
procedure(arg1,arg2,...)/*m*

Execute the named procedure with the given arguments. Arguments can be integer, character or string constants or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to **d**. This facility is only available if the program was loaded with the **–g** option.

linenumber **b** *commands*

Set a breakpoint at the given line. If a procedure name without a line number is given (e.g., "proc:"), a breakpoint is placed at the first line in the procedure even if it was not compiled with the **–g** option. If no *linenumber* is given, a breakpoint is placed at the current line. If no *commands* are given, execution stops just before the breakpoint and control is returned to *sdb*. Otherwise, the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If *k* is used as a command to execute at a breakpoint, control returns to *sdb*, instead of continuing execution.

B

Print a list of the currently active breakpoints.

linenumber **d**

Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d** then the breakpoint is deleted.

D

Delete all breakpoints.

l
Print the last executed line.

linenumber **a**
Announce. If *linenumber* is of the form *proc*:*number*, the command effectively does a *linenumber* **b l**. If *linenumber* is of the form *proc*:, the command effectively does a *proc*: **b T**.

Miscellaneous commands:

!*command*
The command is interpreted by *sh*(1).

newline
If the previous command printed a source line, then advance the current line by one line and print the new current line. If the previous command displayed a memory location, then display the next memory location.

end-of-file character
Scroll. Print the next 10 lines of instructions, source or data depending on which was printed last. The end-of-file character is usually CTRL-D.

< *filename*
Read commands from *file name* until the end of file is reached, and then continue to accept commands from standard input. When *sdb* is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

M
Print the address maps.

M [?/] [*] *b e f*
Record new values for the address map. The arguments ? and / specify the text and data maps, respectively. The first segment (*b1*, *e1*, *f1*) is changed unless * is specified, in which case the second segment (*b2*, *e2*, *f2*) of the mapping is changed. If fewer than three values are given, the remaining map parameters are left unchanged.

" *string*
Print the given string. The C escape sequences of the form *character* are recognized, where *character* is a nonnumeric character.

q
Exit the debugger.

h
Print help.

SDBPS1
The shell environment variable $SDBPS1 if set, will be used as the *sdb* user prompt in place of the default asterisk prompt.

The following commands also exist and are intended only for debugging the debugger:

V
Print the version number.

Q
Print a list of procedures and files being debugged.

FILES

a.out
core

SEE ALSO

cc(1), a.out(4), core(4), syms(4)
sh(1) in the *User's Reference Manual.*

WARNINGS

When *sdb* prints the value of an external variable for which there is no debugging information, a warning is printed before the value. The size is assumed to be **int** (integer).

Line number information in optimized functions is unreliable, the compiler always performs some minimal optimisation which can re-order line-numbers in the final object code.

If a procedure modifies any of its arguments, *sdb* will always print the modified value rather than the original, this can lead to misleading stack back-traces.

BUGS

Current compiler optimisation causes problems when printing local variables, the effects may be reduced with the Greenhills compiler by using the **–X159** option which disables some of the register optimisation used by the GreenHills compiler. The *cc* front-end automatically passes the -X159 option to the compiler when it detects the **-g** option on the command line.

Even when the -X159 option is used the debugger may still have trouble locating some automatic variables when tracing programs under *sdb*, care should be exercised when attempting to modify local (automatic) variables.

Stack back-tracing on the 88100 causes problems due to the lack of a frame pointer register. The stack back-trace code relies heavily on code examination to perform stack frame construction and register value location during code tracing. Certain code modules, especially those built from assembly code rather than 'C' source can cause the back-trace to fail. This results in problems ranging from incorrect data symbols in disassembled code to total failure to provide a back-trace.

If a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a core image.

NAME

sdiff – side-by-side difference program

SYNOPSIS

sdiff [*options* ...] *file1 file2*

DESCRIPTION

sdiff uses the output of *diff*(1) to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files is printed with a blank gutter between them if the lines are identical, a < in the gutter if the line only exists in *file1*, a > in the gutter if the line only exists in *file2*, and a | for lines that are different.

For example:

```
x       |       y
a               a
b       <
c       <
d               d
        >       c
```

The following options exist:

–w *n*
Use the next argument, *n*, as the width of the output line. The default line length is 130 characters.

–l
Only print the left side of any lines that are identical.

–s
Do not print identical lines.

–o *output*
Use the next argument, *output*, as 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*. Sets of differences, as produced by *diff*(1), are printed; where a set of differences share a common gutter character. After printing each set of differences, *sdiff* prompts the user with a **%** and waits for one of the following user-typed commands:

l	append the left column to the output file
r	append the right column to the output file

s	turn on silent mode; do not print identical lines
v	turn off silent mode
e l	call the editor with the left column
e r	call the editor with the right column
e b	call the editor with the concatenation of left and right
e	call the editor with a zero length file
q	exit from the program

On exit from the editor, the resulting file is concatenated on the end of the *output* file.

SEE ALSO

diff(1), ed(1)

NAME

sed – stream editor

SYNOPSIS

sed [**–n**] [**–e** *script*] [**–f** *sfilef*] [*tfiles*]

DESCRIPTION

sed copies the named *files* (standard input default) to the standard output, edited according to a script of commands. The **–f** option causes the script to be taken from file *sfile*; these options accumulate. If there is just one **–e** option and no **–f** options, the flag **–e** may be omitted. The **–n** option suppresses the default output. A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation, *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a **D** command), applies in sequence all commands whose *addresses* select that pattern space, and at the end of the script copies the pattern space to the standard output (except under **–n**) and deletes the pattern space.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval.

An *address* is either a decimal number that counts input lines cumulatively across files, a **$** that addresses the last line of input, or a context address, i.e., a */regular expression/* in the style of *ed*(1) modified thus:

In a context address, the construction *****?*regular expression*?, where ? is any character, is identical to */regular expression/*. Note that in the context address **\xabc\xdefx**, the second **x** stands for itself, so that the regular expression is **abcxdef**.

The escape sequence **\n** matches a newline *embedded* in the pattern space.

A period **.** matches any character except the *terminal* newline of the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.)Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function **!** (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

The *text* argument consists of one or more lines, all but the last of which end with **** to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an **s** command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line. The *rfile* or *wfile* argument must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 10 distinct *wfile* arguments.

(1) **a**
text Append. Place *text* on the output before reading the next input line.

(2) **b** *label* Branch to the **:** command bearing the *label*. If *label* is empty, branch to the end of the script.

(2) **c**
text Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.

(2) **d** Delete the pattern space. Start the next cycle.

(2) **D** Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2) **g** Replace the contents of the pattern space by the contents of the hold space.

(2) **G** Append the contents of the hold space to the pattern space.

(2) **h** Replace the contents of the hold space by the contents of the pattern space.

(2) **H** Append the contents of the pattern space to the hold space.

(1) **i**
text Insert. Place *text* on the standard output.

(2) **l** List the pattern space on the standard output in an unambiguous form. Non-printable characters are displayed in octal notation and long lines are folded.

(2) **n** Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2) **N** Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2) **p** Print. Copy the pattern space to the standard output.

(2) **P** Copy the initial segment of the pattern space through the first newline to the standard output.

(1) **q** Quit. Branch to the end of the script. Do not start a new cycle.

(2) **r** *rfile* Read the contents of *rfile*. Place them on the output before reading the next input line.

(2) **s**/*regular expression*/*replacement*/*flags*

Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character may be used instead of /. For a fuller description see *ed*(1). *Flags* is zero or more of:

- **n** n= 1 - 512. Substitute for just the n th occurrence of the *regular expression*.
- **g** Global. Substitute for all nonoverlapping instances of the *regular expression* rather than just the first one.
- **p** Print the pattern space if a replacement was made.
- **w** *wfile* Write. Append the pattern space to *wfile* if a replacement was made.

(2) **t** *label* Test. Branch to the : command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a **t**. If *label* is empty, branch to the end of the script.

(2) **w** *wfile* Write. Append the pattern space to *wfile*.

(2) **x** Exchange the contents of the pattern and hold spaces.

(2) **y**/*string1*/*string2*/

Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal.

(2)**!** *function*

Don't. Apply the *function* (or group, if *function* is **{**) only to lines *not* selected by the address(es).

(0) : *label* This command does nothing; it bears a *label* for **b** and **t** commands to branch to.

(1) = Place the current line number on the standard output as a line.

(2) { Execute the following commands through a matching } only when the pattern space is selected.

(0) An empty command is ignored.

(0) # If a # appears as the first character on the first line of a script file, then that entire line is treated as a comment, with one exception. If the character after the # is an 'n', then the default output will be suppressed. The rest of the line after #n is also ignored. A script file must contain at least one non-comment line.

SEE ALSO

awk(1), ed(1), grep(1)

NAME

setpgrp – set process group id and execute command

SYNOPSIS

setpgrp *command* [*arg* ...]

DESCRIPTION

The *setpgrp* command sets its own process group id to its process id and then executes the command line that is the argument list.

SEE ALSO

getpgrp(2), setpgrp(2), getpid(2)

NAME

setup – initialize system for first user

SYNOPSIS

setup

DESCRIPTION

The *setup* command, which is also accessible as a login by the same name, allows the first user to be established as the "owner" of the machine.

The user is permitted to add the first logins to the system, usually starting with his or her own.

The user can then protect the system from unauthorized modification of the machine configuration and software by giving passwords to the administrative and maintenance functions. Normally, the first user of the machine enters this command through the setup login, which initially has no password, and then gives passwords to the various functions in the system. Any that the user leaves without password protection can be exercised by anyone.

The user can then give passwords to system logins such as "root", "bin", etc. (*provided they do not already have passwords*). Once given a password, each login can only be changed by that login or "root".

The user can then set the date, time and time zone of the machine.

The user can then set the node name of the machine.

SEE ALSO

passwd(1)

DIAGNOSTICS

The *passwd*(1) command complains if the password provided does not meet its standards.

WARNING

If the setup login is not under password control, anyone can put passwords on the other functions.

NAME

sh, rsh – shell, the standard/restricted command programming language

SYNOPSIS

sh [**–acefhiknrstuvx**] [*args*]
rsh [**–acefhiknrstuvx**] [*args*]

DESCRIPTION

sh is a command programming language that executes commands read from a terminal or a file. *rsh* is a restricted version of the standard command interpreter *sh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See **Invocation** for the meaning of arguments to the shell.

Definitions

A *blank* is a tab or a space. A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a name, a digit, or any of the characters *, @, #, ?, –, $, and !.

Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec*(2)). The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by |. The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ; or &. Of these four symbols, ; and & have equal precedence, which is lower than that of && and ||. The symbols && and || also have equal precedence. A semicolon (;) causes sequential execution of the preceding pipeline; an ampersand (&) causes asynchronous execution of the preceding pipeline (i.e., the shell does *not* wait for that pipeline to finish). The symbol && (||) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of newlines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a *simple-command* or one of the following. Unless otherwise stated, the value returned by a command is that of the last *simple-command* executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**
Each time a **for** command is executed, *name* is set to the next *word* taken from the **in** *word* list. If **in** *word* ... is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see *Parameter Substitution* below). Execution ends when there are no more words in the list.

case *word* **in** [*pattern* [| *pattern*] ...) *list* ;;] ... **esac**
A **case** command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see **File Name Generation**) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns a zero exit status.

while *list* **do** *list* **done**
A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

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

{*list***;}**
list is executed in the current (that is, parent) shell.

name **() {***list***;}**
Define a function which is referenced by *name*. The body of the function is the *list* of commands between **{** and **}**. Execution of functions is described below (see **Execution**).

The following words are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { }

Comments

A word beginning with **#** causes that word and all the following characters up to a newline to be ignored.

Command Substitution

The shell reads commands from the string between two grave accents (`` ` ` ``) and the standard output from these commands may be used as all or part of a word. Trailing newlines from the standard output are removed.

No interpretation is done on the string before the string is read, except to remove backslashes (\) used to escape other characters. Backslashes may be used to escape a grave accent (`` ` ``) or another backslash (\) and are removed before the command string is read. Escaping grave accents allows nested command substitution. If the command substitution lies within a pair of double quotes (" ...` ...` ... "), a backslash used to escape a double quote (\") will be removed; otherwise, it will be left intact.

If a backslash is used to escape a newline character (**\newline**), both the backslash and the newline are removed (see the later section on Quoting). In addition, backslashes used to escape dollar signs (**\$**) are removed. Since no interpretation is done on the command string before it is read, inserting a backslash to escape a dollar sign has no effect. Backslashes that precede characters other than \, `` ` ``, ", **newline**, and **$** are left intact when the command string is read.

Parameter Substitution

The character **$** is used to introduce substitutable *parameters*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Positional parameters may be assigned values by **set**. Keyword parameters (also known as variables) may be assigned values by writing:

name = value [name = value] ...

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

${*parameter*}
The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is * or @, all the positional parameters, starting with **$1**, are substituted (separated by spaces). Parameter **$0** is set from argument zero when the shell is invoked.

${*parameter*:-*word*}
If *parameter* is set and is non-**NULL**, substitute its value; otherwise substitute *word*.

${*parameter*:=*word*}
If *parameter* is not set or is **NULL** set it to *word*; the value of the parameter is substituted. Positional parameters may not be assigned to in this way.

${*parameter*:?*word*}
If *parameter* is set and is non-**NULL**, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, the message "parameter **NULL** or not set" is printed.

${*parameter*:+*word*}
If *parameter* is set and is non-**NULL**, substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if d is not set or is **NULL**:

```
echo ${d:-`pwd`}
```

If the colon (:) is omitted from the above expressions, the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

#
The number of positional parameters in decimal.

-
Flags supplied to the shell on invocation or by the **set** command.

?
The decimal value returned by the last synchronously executed command.

$
The process number of this shell.

!

The process number of the last background command invoked.

The following parameters are used by the shell:

HOME

The default argument (home directory) for the *cd* command.

PATH

The search path for commands (see **Execution**). The user may not change PATH if executing under *rsh*.

CDPATH

The search path for the *cd* command.

MAIL

If this parameter is set to the name of a mail file *and* the MAILPATH parameter is not set, the shell informs the user of the arrival of mail in the specified file.

MAILCHECK

This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the MAILPATH or MAIL parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.

MAILPATH

A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by **%** and a message that will be printed when the modification time changes. The default message is **`you have mail`**.

PS1

Primary prompt string, by default "**$** ".

PS2

Secondary prompt string, by default "**>** ".

IFS

Internal field separators, normally **space**, **tab**, and **newline**.

SHACCT

If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed.

SHELL
When the shell is invoked, it scans the environment (see **Environment**) for this name. If it is found and 'rsh' is the file name part of its value, the shell becomes a restricted shell.

The shell gives default values to PATH, PS1, PS2, MAILCHECK and IFS. **HOME** and **MAIL** are set by *login*(1).

Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (those found in IFS) and split into distinct arguments where such characters are found. Explicit NULL arguments ("" or '') are retained. Implicit NULL arguments (those resulting from *parameters* that have no values) are removed.

Input/Output

A command's input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a *simple-command* or may precede or follow a *command* and are *not* passed on as arguments to the invoked command. Note that parameter and command substitution occurs before *word* or *digit* is used.

<word
Use file *word* as standard input (file descriptor 0).

>word
Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.

>>word
Use file *word* as standard output. If the file exists, output is appended to it (by first seeking to the EOF); otherwise, the file is created.

<<[-]word
After parameter and command substitution is done on *word*, the shell input is read up to the first line that literally matches the resulting *word*, or to an EOF. If, however, - is appended to <<:

1) leading tabs are stripped from *word* before the shell input is read (but after parameter and command substitution is done on *word*),

2) leading tabs are stripped from the shell input as it is read and before each line is compared with *word*, and

3) shell input is read up to the first line that literally matches the resulting *word*, or to an end-of-file.

If any character of *word* is quoted (see **Quoting**), no additional processing is done to the shell input. If no characters of *word* are quoted:

1) parameter and command substitution occurs,
2) (escaped) **\newline** is ignored, and
3) **** must be used to quote the characters ****, **$**, and **`**.

The resulting document becomes the standard input.

<&digit
Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using **>&digit**.

<&-
The standard input is closed. Similarly for the standard output using **>&-**.

If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

... 2>&1

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

... 1>*xxx* 2>&1

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e., *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

Using the terminology introduced on the first page, under **Commands**, if a *command* is composed of several *simple commands*, redirection will be evaluated for the entire *command* before it is evaluated for each *simple command*. That is, the shell evaluates redirection for the entire *list*, then each *pipeline* within the *list*, then each *command* within each *pipeline*, then each *list* within each *command*.

If a command is followed by **&** the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Redirection of output is not allowed in the restricted shell.

File Name Generation

Before a command is executed, each command *word* is scanned for the characters *****, **?**, and **[**. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character **.** at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

***** Matches any string, including the **NULL** string.

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by **–** matches any character lexically between the pair, inclusive. If the first character following the opening ``[´´ is a **"!"** any character not enclosed is matched.

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

; & ( ) | ^ < > newline space tab

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a backslash (\) or inserting it between a pair of quote marks (**´´** or **""**). During processing, the shell may quote certain characters to prevent them from taking on a special meaning. Backslashes used to quote a single character are removed from the word before the command is executed. The pair **\newline** is removed from a word before command and parameter substitution.

All characters enclosed between a pair of single quote marks (''), except a single quote, are quoted by the shell. Backslash has no special meaning inside a pair of single quotes. A single quote may be quoted inside a pair of double quote marks (for example, "'").

Inside a pair of double quote marks (""), parameter and command substitution occurs and the shell quotes the results to avoid blank interpretation and file name generation. If **$*** is within a pair of double quotes, the positional parameters are substituted and quoted, separated by quoted spaces ("**$1 $2** ..."); however, if **$@** is within a pair of double quotes, the positional parameters are substituted and quoted, separated by unquoted spaces ("**$1**" "**$2**" ...). **** quotes the characters ****, **`**, **"**, and **$**. The pair **\newline** is removed before parameter and command substitution. If a backslash precedes characters other than ****, **`**, **"**, **$**, and newline, then the backslash itself is quoted by the shell.

Prompting

When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt (i.e., the value of PS2) is issued.

Environment

The *environment* (see *environ*(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the *export* command is used to bind the shell's parameter to the environment (see also **set -a**). A parameter may be removed from the environment with the **unset** command. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by **unset**, plus any modifications or additions, all of which must be noted in *export* commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. Thus:

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

are equivalent (as far as the execution of *cmd* is concerned).

If the **–k** flag is set, *all* keyword arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and **c**:

```
echo a=b c
set –k
echo a=b c
```

Signals

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by **&**; otherwise, signals have the values inherited by the shell from its parent, with the exception of signal 11 (but see also the *trap* command).

Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the **Special Commands** listed below, it is executed in the shell process. If the command name does not match a **Special Command**, but matches the name of a defined function, the function is executed in the shell process (note how this differs from the execution of shell procedures). The positional parameters **$1**, **$2**, ... are set to the arguments of the function. If the command name matches neither a *Special Command* nor the name of a defined function, a new process is created and an attempt is made to execute the command via *exec*(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is **:/bin:/usr/bin** (specifying the current directory, **/bin**, and **/usr/bin**, in that order). Note that the current directory is specified by a **NULL** pathname, which can appear immediately after the equal sign, between two colon delimiters anywhere in the path list, or at the end of the path list. If the command name contains a / the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. A parenthesized command is also executed in a sub-shell.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *execs* later). If the command was found in a relative directory, its location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the **PATH** variable is changed or the **hash -r** command is executed (see below).

Special Commands

Input/output redirection is now permitted for these commands. File descriptor 1 is the default output location.

:
No effect; the command does nothing. A zero exit code is returned.

. *file*
Read and execute commands from *file* and return. The search path specified by **PATH** is used to find the directory containing *file*.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter **HOME** is the default *arg*. The shell parameter **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is **NULL** (specifying the current directory). Note that the current directory is specified by a NULL pathname, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / the search path is not used. Otherwise, each directory in the path is searched for *arg*. The *cd* command may not be executed by *rsh*.

echo [*arg* ...]
Echo arguments. See *echo*(1) for usage and description.

eval [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*]
Causes a shell to exit with the exit status specified by *n*. If *n* is omitted the exit status is that of the last command executed (an EOF will also cause the shell to exit.)

export [*name* ...]
The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, variable names that have been marked for export during the current shell's execution are listed. (Variable names exported from a parent shell are listed only if they have been exported again during the current shell's execution.) Function names are *not* exported.

getopts
Use in shell scripts to support command syntax standards (see *intro*(1)); it parses positional parameters and checks for legal options. See *getopts*(1) for usage and description.

hash [**–r**] [*name* ...]
For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *hits* is the number of times a command has been invoked by the shell process. *cost* is a measure of the work required to locate a command in the search path. If a command is found in a "relative" directory in the search path, after changing to that directory, the stored location of that command is recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *cost* will be incremented when the recalculation is done.

newgrp [*arg* ...]
Equivalent to **exec newgrp** *arg* .... See *newgrp*(1) for usage and description.

pwd
Print the current working directory. See *pwd*(1) for usage and description.

read [*name* ...]
One line is read from the standard input and, using the internal field separator, **IFS** (normally space or tab), to delimit word boundaries, the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. Lines can be continued using **\newline**. Characters other than **newline** can be quoted by preceding them with a backslash. These backslashes are removed before words are assigned to *names*, and no interpretation is done on the character that follows the backslash. The return code is 0 unless an EOF is encountered.

readonly [*name* ...]
The given *name*s are marked *readonly* and the values of the these *name*s may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed.

return [*n*]
Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

set [**—aefhkntuvx** [*arg* ...]]

–a
Mark variables which are modified or created for export.

–e
Exit immediately if a command exits with a non-zero exit status.

–f
Disable file name generation

–h
Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).

–k
All keyword arguments are placed in the environment for a command, not just those that precede the command name.

–n
Read commands but do not execute them.

–t
Exit after reading and executing one command.

–u
Treat unset variables as an error when substituting.

–v
Print shell input lines as they are read.

–x
Print commands and their arguments as they are executed.

—
Do not change any of the flags; useful in setting **$1** to –.

Using + rather than – causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in **$–**. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given the values of all names are printed.

shift [*n*]
The positional parameters from **$n+1** ... are renamed **$1** .... If *n* is not given, it is assumed to be 1.

test
Evaluate conditional expressions. See *test*(1) for usage and description.

times
Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] . . .
The command *arg* is to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is absent all trap(s) *n* are reset to their original values. If *arg* is the **NULL** string this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *arg* is executed on exit from the shell. The *trap* command with no arguments prints a list of commands associated with each signal number.

type [*name* . . .]
For each *name*, indicate how it would be interpreted if used as a command name.

ulimit [*n*]
Impose a size limit of *n* blocks on files written by the shell and its child processes (files of any size may be read). If *n* is omitted, the current limit is printed. You may lower your own ulimit, but only a super-user (see *su*(1M)) can raise a ulimit.

umask [*nnn*]
The user file-creation mask is set to *nnn* (see *umask*(1)). If *nnn* is omitted, the current value of the mask is printed.

unset [*name* . . .]
For each *name*, remove the corresponding variable or function. The variables **PATH, PS1, PS2, MAILCHECK** and **IFS** cannot be unset.

wait [*n*]
Wait for your background process whose process id is *n* and report its termination status. If *n* is omitted, all your shell's currently active background processes are waited for and the return code will be zero.

Invocation

If the shell is invoked through *exec*(2) and the first character of argument zero is –, commands are initially read from **/etc/profile** and from *$HOME/.profile*, if such files exist. Thereafter, commands are read as described below, which is also the case when the shell is invoked as */bin/sh*. The flags below are interpreted by the shell on invocation only; Note that unless the **–c** or **–s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

–c *string*
If the **–c** flag is present commands are read from *string*.

–s
If the **–s** flag is present or if no arguments remain commands are read from the standard input. Any remaining arguments specify the positional parameters. Shell output (except for *Special Commands*) is written to file descriptor 2.

–i
If the **–i** flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.

–r
If the **–r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the *set* command above.

rsh Only

rsh is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of *rsh* are identical to those of *sh*, except that the following are disallowed:

changing directory (see *cd*(1)),
setting the value of **$PATH**,
specifying path or command names containing /,
redirecting output (> and >>).

The restrictions above are enforced after **.profile** is interpreted.

MOTOROLA INC.

Microcomputer Division
2900 South Diablo Way
Tempe, Arizona 85282
P.O. Box 2953
Phoenix, Arizona 85062

11039 PRINTED IN USA (3/90) WPC 2,500