UNIX Reference Manual
For LMI-Modified System V

2990-0000 (Rev. A)

Published by LMI 1000 Massachusetts Avenue Cambridge MA 02138 USA

This manual contains material from the documents *Administrator's Manual: System V, UNIX* System* and *Transition Aids: System V, UNIX* System*, published by Western Electric; further material has been added by LMI.

Portions of this document were copyrighted:
1979 Bell Telephone Laboratories, Inc.
1980 Western Electric Company, Inc.
1983 Western Electric Company, Inc.

*UNIX™ is a trademark of American Telephone & Telegraph.

LMI Lambda™ is a trademark of LISP Machine Inc.

PDP, VAX, DEC, UNIBUS, MASSBUS, and SBI are trademarks of Digital Equipment Corporation.

Formatted with BoTEX version 1.15 of 9 July 1986 on July 27, 1986.

Copyright © 1986 LISP Machine Incorporated.

1. Introduction

This manual documents the features of UNIX System V, as augmented by LMI. It does not provide a general overview of the UNIX system; for this see "The UNIX Time-sharing System" and "UNIX Programming: Second Edition", in *UNIX System V: Supplementary Documents.*

This manual is divided into six sections, some containing subclasses:

1. Commands and Application Programs: containing
    1. General-purpose commands 1C. Communications commands 1G. Graphics commands
2. System Calls
3. Subroutines, containing
    3C. C and Assembler Library routines
    3F. Fortran Library routines
    3M. Mathematical Library routines
    3S. Standard I/O Library routines
    3X. Miscellaneous routines
4. File Formats
5. Miscellaneous Facilities
6. Games

Commands and Application Programs describes programs intended to be invoked directly by the user or by command language procedures, as opposed to subroutines, which are intended to be called by the user's programs. Commands generally reside in the directory **/bin** (for **bin**ary programs). Some programs also reside in **/usr/bin**, to save space in **/bin**. These directories are searched automatically by the command interpreter called the *shell.* Subclass 1C contains communication programs such as *cu, send, uucp*, etc.

System Calls describes the entries into the UNIX system kernel, including the C language interface.

Subroutines describes the available subroutines. Their binary versions reside in various system libraries in the directories /lib and /usr/lib. See *intro*(3) for descriptions of these libraries and the files in which they are stored.

File Formats documents the structure of particular kinds of files; for example, the format of the output of the link editor is given in *a.out*(4). Excluded are files used by only one command, for example, the assembler's intermediate files.

In general, the C language struct declarations corresponding to these formats can be found in the directories /usr/include and /usr/include/sys.

Miscellaneous Facilities describes a variety of things: character sets, macro packages, etc.

Games describes the games and educational programs that reside in the directory **/usr/games**.

Each section consists of a number of independent entries of a page or so each. The name of the entry appears in the upper corners of its pages. Entries within each section are alphabetized, with the exception of the introductory entry that begins each section. The page numbers of each entry start at 1. Some entries may describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its "major" name.

All entries are based on a common format, not all of whose parts always appear.

- The **Name** section gives the name(s) of the entry and briefly states its purpose.
- The **Synopsis** summarizes the use of the program being described. A few conventions are used, particularly in Section 1 (Commands):

  Boldface strings are literals and are to be entered just as they appear.

  Italic strings generally represent substitutable argument prototypes and program names found elsewhere in the manual.

  Square brackets ([ ]) around an argument prototype indicate that the argument is optional. When an argument prototype is given as "name" or "file", it always refers to a filename.

  Ellipses (. . .) are used to show that the previous argument prototype may be repeated.

  A final convention is used by the commands themselves. An argument beginning with a minus (-), plus (+), or equal sign (=) is often taken to be some sort of flag argument, even if it appears in a position where a filename could appear. Therefore it is unwise to have files whose names begin with -, +, or =.
- The **Description** discusses the subject at hand.
- The **Example(s)** part gives example(s) of usage where appropriate.
- **See Also** gives pointers to related information.
- **Diagnostics** discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.
- **Warnings** points out potential pitfalls.

- The **Bugs** section describes known bugs and deficiencies, if any. Occasionally, a fix is also suggested.

A table of contents to these sections precedes them; a permuted index follows them. On each index line, the title of the entry to which that line refers is followed by the appropriate section number in parentheses. This is important because there is considerable duplication of names among the sections, arising principally from commands that exist only to exercise a particular system call.

On most systems, all entries are available online via the *man*(1) command.

TABLE OF CONTENTS

1. Commands and Application Programs

2. System Calls

3. Subroutines

4. File Formats

5. Miscellaneous Facilities

6. Games

NAME

intro – introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, publicly-accessible commands. Certain distinctions of purpose are made in the headings:

(1) Commands of general utility.
(1C) Commands for communication with other systems.
(1G) Commands used primarily for graphics and computer-aided design.

COMMAND SYNTAX

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

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

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

SEE ALSO

getopt(1), getopt(3C).
Section 6 of this volume for computer games.
The "How to Get Started" section in the Introduction to this manual.

DIAGNOSTICS

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

BUGS

Regretfully, many commands do not adhere to the aforementioned syntax.

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

300 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. It also attempts to draw Greek letters and other special symbols. It permits convenient use of 12-pitch text. It also reduces printing time 5% to 70%. *300* can be used to print equations neatly, in the sequence:

neqn file ... | nroff | 300

WARNING: if your terminal has a PLOT switch, make sure it is turned *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/inch text. DASI 300 terminals normally allow only two combinations: 10-pitch, 6 lines/inch, or 12-pitch, 8 lines/inch. To obtain the 12-pitch, 6 lines per inch combination, 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 line-feed requires 8 increments, while a 12-pitch line-feed needs only 6. The first digit of *n* overrides the default value, thus allowing 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**. The user could also obtain appropriate half-lines for 12-pitch, 8 lines/inch mode by using the option **– 3** alone, having set 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 nulls 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 print properly otherwise. 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 line-feed delays. The *stty*(1) modes **nl0 cr2** or **nl0 cr3** are recommended for most uses.

300 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 line-feed 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

Thus, the use of *300* can 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.

The *neqn* names of, and resulting output for, the Greek and special characters supported by *300* are shown in *greek*(5).

SEE ALSO

450(1), eqn(1), graph(1G), mesg(1), nroff(1), stty(1), tabs(1), tbl(1), tplot(1G), greek(5).

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 line-feeds, 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 line-feeds.

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 Teletypewriter 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* sends the *cmd* to the shell.

The command line options are:

– **t** Don't 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* Set page length to *L*; *L* accepts the scale factors **i** (inches) and **l** (lines); default is lines.

SEE ALSO

pr(1), tc(1), troff(1).

NAME

450 – handle special functions of the DASI 450 terminal

SYNOPSIS

450

DESCRIPTION

450 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). *450* can be used to print equations neatly, in the sequence:

neqn file ... | nroff | 450

WARNING: Make sure that 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 lines/inch, unless dynamically changed to 8 lines per inch by an appropriate escape sequence.

450 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 line-feed key to get any response.

In many (but not all) cases, the use of *450* can be replaced by one of the following:

nroff – T450 files ...

or

nroff – T450– 12 files ...

Thus, the use of *450* can 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), eqn(1), graph(1G), mesg(1), nroff(1), stty(1), tabs(1), tbl(1), tplot(1G), greek(5).

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 line-feeds, 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 line-feeds.

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 (the *fork/exec* flag: 1 for *fork* without *exec*) and STAT (the system exit status).

The command name is prepended with a # if it 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:

– b	Read backwards, showing latest commands first.
– 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).
– 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 ? which designates only those processes associated with unknown user IDs.
– g *group*	Show only processes belonging to *group*. The *group* may be designated by either the group ID or group name.
– d *mm/dd*	Any *time* arguments following this flag are assumed to occur on the given month *mm* and the day *dd* rather than during the last 24 hours. This is needed for looking at old files.
– s *time*	Select processes existing at or after *time*, given in the format *hr* [:*min* [:*sec*]].
– e *time*	Select processes existing at or before *time*.
– S *time*	Select processes starting at or after *time*.
– E *time*	Select processes ending at or before *time*.
– n *pattern*	Show only commands matching *pattern* that may be a regular expression as in *ed*(1) except that + means one or more occurrences.
– 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.

Listing options together has the effect of a logical *and*.

FILES

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

SEE ALSO

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

BUGS

Acctcom only reports on processes that have terminated; use *ps*(1) for active processes. If *time* exceeds the present time and option **-d** is not used, then *time* is interpreted as occurring on the previous day.

NAME

admin – create and administer SCCS files

SYNOPSIS

admin [– n] [– i[name]] [– rrel] [– t[name]] [– fflag[flag-val]] [– dflag[flag-val]] [– alogin] [– elogin] [– 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 filenames must begin with the characters **s.**). If a named file doesn't 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 keyletter arguments are as follows. 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 that a new SCCS file is to be created.

– 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 filename is omitted, the text is obtained by reading the standard input until an end-of-file 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. Use of 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 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 (SID) to be used by a *get*(1) command.

i 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.

j Allows concurrent *get*(1) commands for editing on the same SID 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> ::= *RELEASE NUMBER* | **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 UNIX System group ID, to be added to the list of users who 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 *logins*, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add 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.

– 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).

– m[*mrlist*] The list of Modification Requests (*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.

– h Causes *admin* to check the structure of the SCCS file (see *sccsfile*(5)), and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This 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 check-sum 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.

FILES

The last component of all SCCS filenames must be of the form **s.***filename*. 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.***filename*, (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.

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1), sccsfile(4).
"Source Code Control System User's Guide" in the *User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

apropos – locate commands by keyword lookup

SYNOPSIS

apropos keyword ...

DESCRIPTION

Apropos shows which manual sections contain instances of any of the given keywords in their title. Each word is considered separately and case of letters is ignored. Words which are part of other words are considered thus looking for compile will hit all instances of 'compiler' also. Try

apropos password

and

apropos editor

If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. Try 'apropos format' and then 'man 3s printf' to get the manual on the subroutine *printf*.

Apropos is actually just the **–k** option to the *man*(1) command.

FILES

/usr/man/whatis data base

SEE ALSO

man(1), whatis(1).

NAME

ar – archive and library maintainer for portable archives

SYNOPSIS

ar key [posname] afile name ...

DESCRIPTION

Ar 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.

When *ar* creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure are described in detail in *ar*(4). The archive symbol table (described in *ar*(4)) is used by the link editor (*ld*(1)) to effect multiple passes over libraries of object files in an efficient manner. Whenever the *ar*(1) command is used to create or update the contents of an archive, the symbol table is rebuilt. The symbol table can be forced to be rebuilt by the **s** option described below.

Key is one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcls**. *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 modified dates later than the archive files are replaced. If an optional positioning character from the set **abi** is used, then 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 whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

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, then 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.

v Verbose. Under the verbose option, *ar* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, it gives a long listing of all information about the files. When used with **x**, it precedes each file with a name.

c Create. Normally *ar* creates *afile* when it needs to. The create option suppresses the normal message that is produced when *afile* is created.

l Local. Normally *ar* places its temporary files in the directory **/tmp**. This option causes them to be placed in the local directory.

s Symbol table creation. 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

/tmp/ar* temporaries

SEE ALSO

ld(1), lorder(1), strip(1), a.out(4), ar(4).

BUGS

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

NAME

as, ljas – common assembler

SYNOPSIS

as [– **o** objfile] [– **n**] [– **m**] [– **R**] [– **V**] filename
ljas [– **o** objfile] [– **n**] [– **m**] [– **R**] [– **V**] filename

DESCRIPTION

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

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

– **n** Turn off long/short address optimization. By default, address optimization takes place.

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

– **R** Remove (unlink) the input file after assembly is completed. This option is off by default.

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

The *ljas* command is a special version of the *as* assembler. *Ljas* produces "long jump" instructions rather than (short) branch instructions.

FILES

/usr/tmp/as[1-6]*XXXXXX* temporary files

SEE ALSO

ld(1), m4(1), nm(1), strip(1), a.out(4).

WARNING

If the – **m** (*m4* macro pre-processor invocation) option is used, keywords for *m4* (see *m4*(1)) cannot be used as symbols (variables, functions, labels) in the input file because *m4* cannot determine which are assembler symbols and which are real *m4* macros.

BUGS

Arithmetic expressions are permitted to have only one forward referenced symbol per expression.

NAME

asa – interpret ASA carriage control characters

SYNOPSIS

asa [files]

DESCRIPTION

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

' '	(blank) single new line before printing
0	double new line 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 ' '. The first character of a line is *not* printed. If any such lines appear, an appropriate diagnostic will appear on standard error. This program forces the first line of each input file to start on a new page.

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

a.out | asa | lpr

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

asa file

SEE ALSO

efl(1), f77(1), fsplit(1), ratfor(1).

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [– Fc] [prog] [parameters] [files]

DESCRIPTION

Awk 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 is 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 *awk*.

Files are read in order; if there are no files, the standard input is read. The filename – 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, as shown below). The fields are denoted **$1**, **$2**, ...; **$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, new-lines, 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 *>expr* is present), 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)).

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 and are 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 new-line); 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: awk – f program n=5 input

SEE ALSO

grep(1), lex(1), sed(1).

"Awk - A Pattern Scanning and Processing Language" by A. V. Aho, B. W. Kernighan, and P. J. Weinberger.

"The 'Awk' Programming Language" in the *Support Tools 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

banner – print large banner on printer

SYNOPSIS

/usr/ucb/banner [**– w***n*] message ...

DESCRIPTION

Banner prints a large, high quality banner on the standard output. If the message is omitted, it prompts for and reads one line of its standard input. If **– w** is given, the output is scrunched down from a width of 132 to *n* , suitable for a narrow terminal. If *n* is omitted, it defaults to 80.

The output should be printed on a hard-copy device, up to 132 columns wide, with no breaks between the pages. The volume is enough that you want a printer or a fast hardcopy terminal, but if you are patient, a decwriter or other 300 baud terminal will do.

BUGS

Several ASCII characters are not defined, notably <, >, [,], \, ^, _, {, }, | and ~. Also, the characters ", ', and & are funny looking (but in a useful way.)

The **– w** option is implemented by skipping some rows and columns. The smaller it gets, the grainier the output. Sometimes it runs letters together.

AUTHOR

Mark Horton

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 sets the shell variable NAME to /usr/src/cmd.

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

SEE ALSO

sh(1).

BUGS

The *basename* of / is null and is considered an error.

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 **-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;
++ -- (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.

Bc is actually a preprocessor for *dc*(1), which it invokes automatically, unless the **-c** (compile only) option is present. In this case the *dc* input is sent to the standard output instead.

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).
"BC - An Arbitrary Precision Desk-Calculator Language" by L. L. Cherry and R. Morris.
"Arbitrary Precision Desk Calculator Language (BC)" in the *Support Tools Guide*.

BUGS

No &&, || yet.
For statement must have all three E's.
Quit is interpreted when read, not when executed.

NAME

bdiff – file comparator for large files

SYNOPSIS

bdiff file1 file2 [n] [– **s**]

DESCRIPTION

Bdiff is used in a manner analogous to *diff*(1) to identify lines that must be changed in two files to bring them into agreement. Its purpose is to allow processing of files that are too large for *diff*. *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. 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. If *file1* (*file2*) is – , the standard input is read. The optional – **s** (silent) argument specifies that no diagnostics are to be printed by *bdiff*. Note, however, that this does not suppress possible exclamations by *diff*. If both optional arguments are specified, they must appear in the order indicated above.

The output of *bdiff* is the same as that of *diff*, with line numbers adjusted to account for the segmenting of the files (that is, 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).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

bfs – big file scanner

SYNOPSIS

bfs [–] name

DESCRIPTION

Bfs is similar to *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 255 characters per line. *Bfs* is usually more efficient than *ed* 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*. 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* are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?. The symbol > indicates downward search without wrap-around; < indicates upward search without wrap-around. Since *bfs* uses a different regular expression-matching routine than *ed*, the regular expressions accepted are slightly wider in scope (see *regcmp*(3X)). 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**, **n**, **p**, **q**, **w**, =, **!** and null commands operate as described under *ed*. 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 end-of-file is reached, an interrupt signal is received, or an error occurs, reading resumes with the file containing the **xf**. Xf commands may be nested to a depth of 10.

xo [*file*]

Further output from the **p** and null commands is diverted to the named *file*, which, if necessary, is created mode 666. 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 new-line, 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 doesn't 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 doesn't issue an error message on bad addresses, so it may be

used to test whether addresses are bad before other commands are executed. Note that the command

```
xb/^/ label
```

is an unconditional jump.
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**. **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. **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**, the following commands all print the first 100 lines:

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

The command

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

produces a global search for the characters **100** and prints each line containing a match. To escape the special meaning of **%** a **** must precede it.

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

can 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 UNIX System command can be stored into a variable. The only requirement is that the first character of *value* be an **!**. For example, the lines

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

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 test the last saved *return code* from the execution of a System V command (**!***command*) or nonzero value, respectively, to the specified label. The two examples below 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 isn't. 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), regcmp(3X).

DIAGNOSTICS

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

NAME

bs – a compiler/interpreter for modest-sized programs

SYNOPSIS

bs [file [args]]

DESCRIPTION

Bs is a remote descendant of Basic and Snobol4 with some C language added. *Bs* is designed for programming tasks where program development time is as important as the resulting speed of execution. Formalities of data declaration and file/process manipulation are minimized. Line-at-a-time debugging, the *trace* and *dump* statements, and useful run-time error messages all simplify program testing. Furthermore, incomplete programs can be debugged; *inner* functions can be tested before *outer* functions have been written and vice versa.

If the command line *file* argument is provided, the file is used for input before the console is read. By default, statements read from the file argument are compiled for later execution. Likewise, statements entered from the console are normally executed immediately (see *compile* and *execute* below). Unless the final operation is assignment, the result of an immediate expression statement is printed.

Bs programs are made up of input lines. If the last character on a line is a \ , the line is continued. *Bs* accepts lines of the following form:

statement
label statement

A label is a *name* (see below) followed by a colon. A label and a variable can have the same name.

A *bs* statement is either an expression or a keyword followed by zero or more expressions. Some keywords (*clear*, *compile*, *!*, *execute*, *include*, *ibase*, *obase*, and *run*) are always executed as they are compiled.

Statement Syntax:

expression
The expression is executed for its side effects (value, assignment or function call). The details of expressions follow the description of statement types below.

break
Break exits from the innermost *for/while* loop.

clear
Clear is executed immediately. It clears the symbol table and compiled statements.

compile [expression]
Succeeding statements are compiled (overrides the immediate execution default). The optional expression is evaluated and used as a filename for further input. A *clear* is associated with this latter case. *Compile* is executed immediately.

continue
Continue transfers to the loop-continuation of the current *for/while* loop.

dump [name]
The name and current value of every non-local variable is printed. Optionally, only the named variable is reported. After an error or interrupt, the number of the last statement and (possibly) the user-function trace are displayed.

exit [expression]
Return to system level. The expression is returned as process status.

execute
Change to immediate execution mode (an interrupt has a similar effect). This statement does not cause stored statements to execute (see *run* below).

for name = expression expression statement
for name = expression expression
...
next

for expression , expression , expression statement
for expression , expression , expression
...
next
The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression. The third and fourth forms require three expressions separated by commas. The first of these is the initialization, the second is the test (true to continue), and the third is the loop-continuation action (normally an increment).

fun f([a, ...]) [v, ...]
...
nuf
Fun defines the function name, arguments, and local variables for a user-written function. Up to ten arguments and local variables are allowed. Such names cannot be arrays, nor can they be I/O associated. Function definitions may not be nested.

freturn
A way to signal the failure of a user-written function. See the interrogation operator (**?**) below. If interrogation is not present, *freturn* merely returns zero. When interrogation *is* active, *freturn* transfers to that expression (possibly by-passing intermediate function returns).

goto name
Control is passed to the internally stored statement with the matching label.

ibase *N*
Ibase sets the input base (radix) to *N*. The only supported values for *N* are **8**, **10** (the default), and **16**. Hexadecimal values 10– 15 are entered as **a**– **f**. A leading digit is required (i.e., **f0a** must be entered as **0f0a**). *Ibase* (and *obase*, below) are executed immediately.

if expression statement
if expression
...
[**else**
...]
fi
The statement (first form) or group of statements (second form) is executed if the expression evaluates to non-zero. The strings **0** and "" (null) evaluate as zero. In the second form, an optional *else* allows for a group of statements to be executed when the first group is not. The only statement permitted on the same line with an *else* is an *if*; only other *fi*'s can be on the same line with a *fi*. The elision of *else* and *if* into an *elif* is supported. Only a single *fi* is required to close an *if* ... *elif* ... [*else* ...] sequence.

include expression
The expression must evaluate to a filename. The file must contain *bs* source statements.

Such statements become part of the program being compiled. *Include* statements may not be nested.

obase *N*
Obase sets the output base to *N* (see *ibase* above).

onintr label
onintr
The *onintr* command provides program control of interrupts. In the first form, control passes to the label given, just as if a *goto* had been executed at the time *onintr* was executed. The effect of the statement is cleared after each interrupt. In the second form, an interrupt causes *bs* to terminate.

return [expression]
The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

run
The random number generator is reset. Control is passed to the first internal statement. If the *run* statement is contained in a file, it should be the last statement.

stop
Execution of internal statements is stopped. *Bs* reverts to immediate mode.

trace [expression]
The *trace* statement controls function tracing. If the expression is null (or evaluates to zero), tracing is turned off; otherwise, a record of user-function calls/returns is printed. Each *return* decrements the *trace* expression value.

while expression statement
while expression
...
next
While is similar to *for* except that only the conditional expression for loop-continuation is given.

! shell command
An immediate escape to the shell.

...
This statement is ignored. It is used to interject commentary in a program.

Expression Syntax:

name
A name is used to specify a variable. Names are composed of a letter (upper or lower case) optionally followed by letters and digits. Only the first six characters of a name are significant. Except for names declared in *fun* statements, all names are global to the program. Names can take on numeric (double float) values, string values, or can be associated with input/output (see the built-in function *open*( ) below).

name ([expression [, expression] ...])
Functions can be called by a name followed by the arguments in parentheses separated by commas. Except for built-in functions (listed below), the name must be defined with a *fun* statement. Arguments to functions are passed by value.

name [expression [, expression] ...]
This syntax is used to reference either arrays or tables (see built-in *table* functions below). For arrays, each expression is truncated to an integer and used as a specifier for the name. The resulting array reference is syntactically identical to a name; **a[1,2]** is the same as **a[1][2]**. The truncated expressions are restricted to values between 0 and 32767.

number
A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an **e** followed by a possibly signed exponent.

string
Character strings are delimited by " characters. The \ escape character allows the double quote (**\"**), new-line (**\n**), carriage return (**\r**), backspace (**\b**), and tab (**\t**) characters to appear in a string. Otherwise, \ stands for itself.

(expression)
Parentheses are used to alter the normal order of evaluation.

(expression, expression [, expression ...]) [expression]
The bracketed expression is used as a subscript to select a comma-separated expression from the parenthesized list. List elements are numbered from the left, starting at zero. The expression:

(False, True)[a == b]

has the value **True** if the comparison is true.

? expression
The interrogation operator tests for the success of the expression rather than its value. It is useful for testing end-of-file (see examples in the *Programming Tips* section below), the result of the *eval* built-in function, and for checking the return from user-written functions (see *freturn*). An interrogation "trap" (e.g., end-of-file) causes an immediate transfer to the most recent interrogation, possibly skipping assignment statements or intervening function levels.

– expression
The result is the negation of the expression.

++ name
Increments the value of the variable (or array reference). The result is the new value.

-- name
Decrements the value of the variable. The result is the new value.

! expression
The logical negation of the expression. Watch out for the shell escape command.

expression *operator* expression
Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. Except for the assignment, concatenation, and relational operators, both operands are converted to numeric form before the function is applied.

Binary Operators (in increasing precedence):

=
= is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left; all other operators bind left to right.

_
_ (underscore) is the concatenation operator.

& |
& (logical and) has result zero if either of its arguments are zero. It has result one if both of its arguments are non-zero; | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments is non-zero. Both operators treat a null string as a zero.

< <= > >= == !=

The relational operators (<, less than; <=, less than or equal; >, greater than; >=, greater than or equal; ==, equal to; !=, not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: **a>b>c** is the same as **a>b & b>c**. A string comparison is made if both operands are strings.

+ -

Add and subtract.

*** / %**

Multiply, divide, and remainder.

^

Exponentiation.

Built-in Functions:

Dealing with arguments

arg(i)

is the value of the *i*-th actual parameter on the current level of function call. At level zero, *arg* returns the *i*-th command-line argument (*arg*(0) returns **bs**).

narg( )

returns the number of arguments passed. At level zero, the command argument count is returned.

Mathematical

abs(x)

is the absolute value of x.

atan(x)

is the arctangent of x. Its value is between $-\pi/2$ and $\pi/2$.

ceil(x)

returns the smallest integer not less than x.

cos(x)

is the cosine of x (radians).

exp(x)

is the exponential function of x.

floor(x)

returns the largest integer not greater than x.

log(x)

is the natural logarithm of x.

rand( )

is a uniformly distributed random number between zero and one.

sin(x)

is the sine of x (radians).

sqrt(x)

is the square root of x.

String operations

size(s)

returns the size (length in bytes) of s.

format(f, a)
returns the formatted value of *a*. *F* is assumed to be a format specification in the style of *printf*(3S). Only the %...**f**, %...**e**, and %...**s** types are safe.

index(x, y)
returns the number of the first position in *x* that any of the characters from *y* matches. No match yields zero.

trans(s, f, t)
Translates characters of the source *s* from matching characters in *f* to a character in the same position in *t*. Source characters that do not appear in *f* are copied to the result. If the string *f* is longer than *t*, source characters that match in the excess portion of *f* do not appear in the result.

substr(s, start, width)
returns the sub-string of *s* defined by the *start*ing position and *width*.

match(string, pattern)
mstring(n)
The *pattern* is similar to the regular expression syntax of the *ed*(1) command. The characters ., [,], ^ (inside brackets), * and $ are special. The *mstring* function returns the *n*– th ($1 <= n <= 10$) substring of the subject that occurred between pairs of the pattern symbols \(and \) for the most recent call to *match*. To succeed, patterns must match the beginning of the string (as if all patterns began with ^). The function returns the number of characters matched. For example:

```
match("a123ab123", ".*\([a- z]\)") == 6
mstring(1) == "b"
```

File handling

open(name, file, function)
close(name)
The *name* argument must be a *bs* variable name (passed as a string). For the *open*, the *file* argument may be **1)** a 0 (zero), 1, or 2 representing standard input, output, or error output, respectively, **2)** a string representing a filename, or **3)** a string beginning with an ! representing a command to be executed (via *sh* - *c*). The *function* argument must be either **r** (read), **w** (write), **W** (write without new-line), or **a** (append). After a *close*, the *name* reverts to being an ordinary variable. The initial associations are:

```
open("get", 0, "r")
open("put", 1, "w")
open("puterr", 2, "w")
```

Examples are given in the following section.

access(s, m)
executes *access*(2).

ftype(s)
returns a single character file type indication: **f** for regular file, **p** for FIFO (i.e., named pipe), **d** for directory, **b** for block special, or **c** for character special.

Tables

table(name, size)
A table in *bs* is an associatively accessed, single-dimension array. "Subscripts" (called keys) are strings (numbers are converted). The *name* argument must be a *bs* variable name (passed as a string). The *size* argument sets the minimum number of elements to be allocated. *Bs* prints an error message and stops on table overflow.

item(name, i)

key()

The *item* function accesses table elements sequentially (in normal use, there is no orderly progression of key values). Where the *item* function accesses values, the *key* function accesses the "subscript" of the previous *item* call. The *name* argument should not be quoted. Since exact table sizes are not defined, the interrogation operator should be used to detect end-of-table, for example:

```
table("t", 100)
 ...
# If word contains "party", the following expression adds one
# to the count of that word:
++t[word]
 ...
# To print out the the key/value pairs:
for i = 0, ?(s = item(t, i)), ++i  if key()  put = key()_":"_s
```

iskey(name, word)

tests whether the key **word** exists in the table **name** and returns one for true, zero for false.

Odds and ends

eval(s)

evaluates the string argument as a *bs* expression. The function is handy for converting numeric strings to numeric internal form. *Eval* can also be used as a crude form of indirection, as in:

```
name = "xyz"
eval("++"_name)
```

which increments the variable *xyz*. In addition, *eval* preceded by the interrogation operator permits the user to control *bs* error conditions. For example:

```
?eval("open(\"X\", \"XXX\", \"r\")")
```

returns the value zero if there is no file named "XXX" (instead of halting the user's program). The following executes a *goto* to the label *L* (if it exists):

```
label="L"
if !(?eval("goto "_label)) puterr = "no label"
```

plot(request, args)

produces output on devices recognized by *tplot*(1G). The *requests* are as follows:

Call	*Function*
plot(0, term)	causes further *plot* output to be piped into *tplot*(1G) with an argument of −T*term*.
plot(4)	"erases" the plotter.
plot(2, string)	labels the current point with *string*.
plot(3, x1, y1, x2, y2)	draws the line between (*x1*,*y1*) and (*x2*,*y2*).
plot(4, x, y, r)	draws a circle with center (*x*,*y*) and radius *r*.
plot(5, x1, y1, x2, y2, x3, y3)	draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).
plot(6)	is not implemented.
plot(7, x, y)	makes the current point (*x*,*y*).

plot(8, x, y)	draws a line from the current point to (x,y).
plot(9, x, y)	draws a point at (x,y).
plot(10, string)	sets the line mode to *string*.
plot(11, x1, y1, x2, y2)	makes $(x1,y1)$ the lower left corner of the plotting area and $(x2,y2)$ the upper right corner of the plotting area.
plot(12, x1, y1, x2, y2)	causes subsequent x (y) coordinates to be multiplied by $x1$ ($y1$) and then added to $x2$ ($y2$) before they are plotted. The initial scaling is **plot(12, 1.0, 1.0, 0.0, 0.0)**.

Some requests do not apply to all plotters. All requests except zero and twelve are implemented by piping characters to *tplot*(1G). See *plot*(4) for more details.

last( )
in immediate mode, returns the most recently computed value.

PROGRAMMING TIPS

Using *bs* as a calculator:

```
$ bs
#   Distance (inches) light travels in a nanosecond.
186000 * 5280 * 12 / 1e9
11.78496
...

#   Compound interest (6% for 5 years on $1,000).
int = .06 / 4
bal = 1000
for i = 1 5*4  bal = bal +  bal*int
bal -  1000
346.855007
...
exit
```

The outline of a typical *bs* program:

```
# initialize things:
var1 = 1
open("read", "infile", "r")
...
# compute:
while  ?(str = read)
        ...
next
# clean up:
close("read")
...
# last statement executed (exit or stop):
exit
# last input line:
run
```

Input/Output examples:

```
#   Copy "oldfile" to "newfile".
open("read", "oldfile", "r")
open("write", "newfile", "w")
```

```
...
while ?(write == read)
...
# close "read" and "write":
close("read")
close("write")

#  . Pipe between commands.
open("ls", "!ls *", "r")
open("pr", "!pr - 2 - h 'List'", "w")
while ?(pr == ls)  ...
...
# be sure to close (wait for) these:
close("ls")
close("pr")
```

SEE ALSO

ed(1), sh(1), tplot(1G), access(2), printf(3S), stdio(3S), plot(4).
See Section 3 of this volume for further description of the mathematical functions (*pow* on *exp*(3M) is used for exponentiation); *bs* uses the Standard Input/Output package.

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. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive. Beware that "cal 78" refers to the early Christian era, not the 20th century.

NAME

calendar – reminder service

SYNOPSIS

calendar [–]

DESCRIPTION

Calendar consults the file **calendar** in a user's current directory and prints out lines containing today's or tomorrow's date. *Calendar* uses *calprog* to figure out today's and tomorrow's dates. The date read by *calprog* may appear anywhere in a line, and most reasonable date representations are recognized, although the month must appear first. For example, "Dec. 7," "december 7," and "12/7" are recognized; "7 Dec" or "seven december" are not. On weekends "tomorrow" extends through Monday. If the *calendar* command is run on a Friday, lines containing the dates for Friday, Saturday, Sunday, and Monday are selected.

When an argument is present, *calendar* does its job for all users who have a file **calendar** in their login directory and sends them any positive results by *mail*(1). Normally this is done daily by facilities in the operating system.

FILES

calendar
/usr/lib/calprog
/etc/passwd
/tmp/cal*

SEE ALSO

mail(1).

BUGS

The **calendar** file must be public information for a user to get reminder service.
Calendar's extended idea of "tomorrow" does not account for holidays.
Numeric dates must be in the form month/day. Separators other than a slash prevent recognition of the date.

NAME

cat – concatenate and print files

SYNOPSIS

cat [**– u**] [**– s**] file ...

DESCRIPTION

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

cat file

prints the file, and:

cat file1 file2 >file3

concatenates the first two files and places the result on the third.

If no input file is given, or if the argument – is encountered, *cat* reads from the standard input file. Output is buffered unless the **– u** option is specified. The **– s** option makes *cat* silent about non-existent files. No input file may be the same as the output file unless it is a special file.

WARNING

Command formats such as

cat file1 file2 >file1

cause the original data in *file1* to be lost. To append file2 to file1, use:

cat file2 >> file1.

SEE ALSO

cp(1), pr(1).

NAME

cat – catenate and print

SYNOPSIS

cat [**–u**] [**–n**] [**–s**] [**–v**] file ...

DESCRIPTION

Cat reads each *file* in sequence and displays it on the standard output. Thus

cat file

displays the file on the standard output, and

cat file1 file2 >file3

concatenates the first two files and places the result on the third.

If no input file is given, or if the argument '– ' is encountered, *cat* reads from the standard input file. Output is buffered in 1024-byte blocks unless the standard output is a terminal, in which case it is line buffered. The **–u** option makes the output completely unbuffered.

The **–n** option displays the output lines preceded by lines numbers, numbered sequentially from 1. Specifying the **–b** option with the **–n** option omits the line numbers from blank lines.

The **–s** option crushes out multiple adjacent empty lines so that the output is displayed single spaced.

The **–v** option displays non-printing characters so that they are visible. Control characters print like ^X for control-x; the delete character (octal 0177) prints as ^?. Non-ascii characters (with the high bit set) are printed as M- (for meta) followed by the character of the low 7 bits. A **–e** option may be given with the **–v** option, which displays a '$' character at the end of each line. Specifying the **–t** option with the **–v** option displays tab characters as ^I.

SEE ALSO

cp(1), ex(1), more(1), pr(1), tail(1)

BUGS

Beware of 'cat a b >a' and 'cat a b >b', which destroy the input files before reading them.

NAME

cb - C program beautifier

SYNOPSIS

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

DESCRIPTION

Cb reads C programs either from its arguments or from the standard input and writes them on the standard output with spacing and indentation that displays the structure of the code. Under default options, *cb* preserves all user new-lines. Under the **-s** flag, *cb* canonicalizes the code to the style of Kernighan and Ritchie in *The C Programming Language*. The **-j** flag causes split lines to be put back together. The **-l** flag causes *cb* to split lines that are longer than *leng*.

SEE ALSO

cc(1).
"The C Programming Language" by B. W. Kernighan and D. M. Ritchie.

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. *Cc* accepts the following types of arguments:

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; however, if a single C program is compiled and loaded all at one go, no **.o** is produced. 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 following flags are interpreted by *cc*. See *ld*(1) for link editor options and *as*(1) for assembler options.

– **c** Suppress the link-editing phase of the compilation, and force an object file to be produced even if only one program is compiled.

– **p** Arrange for the compiler to produce code which counts the number of times each routine is called. Also, if link editing takes place, replace the standard startoff routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program.

– **g** Cause the compiler to generate additional information needed for the use of *sdb*(1).

– **O** Invoke an object-code optimizer. The optimizer moves, merges, and deletes code, so symbolic debugging with line numbers could be confusing when the optimizer is used.

– **W***c,arg1[,arg2...]*

Hand off the argument[s] *argi* to pass *c*, where *c* is one of [**p012al**] indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively. For example:

– **W***a,-m*

Invoke the **m4** macro preprocessor on the input to the assembler. This must be done for a source file that contains assembler escapes.

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

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

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

– **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.

– **U***symbol*

Undefine *symbol* to the preprocessor.

– **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.

– B*string*

Construct pathnames for substitute preprocessor, compiler, assembler, and link editor passes by concatenating *string* with the suffixes **cpp**, **comp**, **optim**, **as**, and **ld**. If *string* is empty it is taken to be **/lib/o**.

Other arguments are taken to be either link editor option arguments or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are link-edited (in the order given) to produce an executable program with the name **a.out** unless the **-o** option of the link editor is used.

The C language standard was extended after UNIX 5.0 to allow arbitrary length variable names. This standard is supported on the M68000 family of processors. The **-T** option causes *cc* to truncate variable names to provide backward compatibility with earlier systems.

FILES

file.c	input file
file.o	object file
file.s	assembly language file
a.out	link-edited output
/usr/tmp/mc68?	temporary
LIBDIR/cpp	preprocessor
LIBDIR/ccom	compiler
LIBDIR/optim	optimizer
BINDIR/as	assembler, *as*(1)
BINDIR/ld	link editor, *ld*(1)
/lib/libc.a	standard library, see (3)

SEE ALSO

as(1), dis(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 *Programming Guide*.

DIAGNOSTICS

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

WARNING

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value are to explicitly call *exit*(2) or to leave the function **main**() with a *return expression;* construct.

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 internal to the shell.

SEE ALSO

pwd(1), sh(1), chdir(2).

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* specified by the – **r** keyletter, of each named SCCS file.

Delta commentary is defined as the Modification Request (MR) and/or 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 path name 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.

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). In order 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 new-line 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 new-line character terminates the *comment* text.

The exact permissions necessary to modify the SCCS file are documented in the "Source Code Control System User's Guide" in the *System V User's Guide*. Simply stated, if you made the delta, you can change its delta commentary; 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.

The same changes can be accomplished with:

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

WARNINGS

If SCCS filenames are supplied to the *cdc* command via the standard input (– on the command line), 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), help(1), prs(1), sccsfile(4).
"Source Code Control System User's Guide" in the *User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

cflow – generate C flow graph

SYNOPSIS

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

DESCRIPTION

Cflow analyzes a collection of C, YACC, LEX, assembler, and object files and attempts to build a graph charting the external references. Files suffixed in **.y** (for YACC), **.l** (for LEX), **.c** (for C), and **.i** are preprocessed (bypassed for **.i** files) as appropriate and then run through the first pass of *lint*(1). (The – **I**, – **D**, and – **U** options of the C-preprocessor are also understood.) Files suffixed with **.s** are assembled and information is extracted (as in **.o** files) from the symbol table. The output of all this non-trivial processing is 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 (line number), followed by a suitable number of tabs indicating the level. These are followed by the name of the global (normally only a function not defined as an external or beginning with an underscore; see below for the – **i** inclusion option), a colon, and the definition of the global. 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 filename 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 file.c
```

produces the 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 **– e** option of *pr*(1) can be used to compress the tab expansion to something less than every eight spaces.

The following options are interpreted by *cflow*:

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

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

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

– dnum Terminate the flow graph at the level specified by the *num* decimal integer. By default this is a very large number. The cutoff depth should not be set to a nonpositive integer.

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), lex(1), lint(1), nm(1), pr(1), yacc(1).

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

cfnt - clear loaded font

SYNOPSIS

cfnt fontnum [window]

DESCRIPTION

Cfnt clears font *fontnum* from window *window*. The font must have already been loaded into the window using **lfnt.** *Fontnum* must be in the range 0 to 6 (7 is the default font and cannot be cleared). If *window* is not supplied it defaults to the window in which the command is being executed in.

SEE ALSO

lfnt(1) lsfnt(1) sfnt(1)

NAME

cftp – Chaosnet file transfer program

SYNOPSIS

cftp [host]

DESCRIPTION

Cftp is the user interface to the Chaosnet file transfer protocol. The program allows a user to transfer files to and from a remote network site.

The client host with which *cftp* is to communicate may be specified on the command line. If this is done, *cftp* will immediately attempt to establish a connection to a FILE server on that host; otherwise, *cftp* will enter its command interpreter and await instructions from the user. When *cftp* is awaiting commands from the user the prompt cftp> is provided the user. The following commands are recognized by *cftp*:

! Invoke a shell on the local machine.

ascii Set the file transfer *type* to network ASCII. Translates to/from the 8-bit LISP machine character set to 7-bit ASCII. This is the most general way to transfer text between unlike systems, and is the default type.

raw Set the file transfer *type* to support character raw transfer. No character set translation occurs.

image Perform super-image mode transfers. Only translates '\n' in the LISP machine character set

bytes Set the file transfer *type* to support 16-bit binary image transfer. Good for transferring press files.

binary Set the file transfer *type* to support logical byte transfer. Logical bytes are transferred from or to naturally packed logical bytes in a short int on the local machine. Logical bytes are packed from left to right within a short int, and do not cross a short int boundary.

bye Terminate the FILE session with the remote server, and return to the command interpreter.

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

dir [*remote-path*]
Print a listing of the files matching the pattern *path*. If *path* is a directory, a wildcard component '*' should be appended in order to list the contents of the directory. If no *path* is specified, *cftp* prompts for a pathname pattern to search.

get [*remote-file*]
Retrieve the *remote-file* and store it on the local machine. If the remote file name is not specified, *cftp* prompts for it. *Cftp* prompts for the local file name. The current setting for transfer mode is used while transferring the file.

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

open [*host*]
Establish a connection to the specified *host* FILE server. An optional port number may be supplied, in which case, *cftp* will attempt to contact a FILE server at that port.

send [*local-file*]
Store a local file on the remote machine. If the local file name is not specified, *cftp*

prompts for it. *Cftp* prompts for the remote file name. File transfer uses the current settings for transfer mode.

quit Terminate the FILE session with the remote server, and exit *cftp*.

status Show the current status of *cftp*.

login [*user-name*]
Identify yourself to the remote FILE server. If the user name is not specified *cftp* prompts for it. *Cftp* always prompts for a password (after disabling local echo), even if the remote machine has no password for the given user. If there is no password, simply type a carriage return.

probe [*remote-file*]
Check the status of a remote file. If no remote file name is specified, then *cftp* will prompt for it.

verbose
Enable verbose mode. In verbose mode, all responses from the FILE server are displayed to the user.

brief Disable verbose mode.

connect [*host*]
A synonym for open.

disconnect
A synonym for bye.

exit A synonym for quit.

? [*command*]
A synonym for help.

NAME

cheval – execute a command on a remote CHAOSnet host

SYNOPSIS

cheval host [***(user** [**passwd**])] command args

DESCRIPTION

Cheval runs *command* on the remote *host*, supplying *args* to the command. By default, *cheval* logs in on the remote host with the username *anonymous*. This may be overridden by supplying a *username* and *password* (if necessary) inside parentheses with a leading '*' immediately preceeding the command. The '*' and parentheses should be quoted to avoid interpretation by the shell.

EXAMPLES

The following *cheval* command will run *cat*(1) on the remote host *snaggle-tooth*. It will copy *snaggle-tooth*'s password file into the local file /tmp/passwd.snag. Notice that the output redirection is interpreted by the local shell.

```
cheval snaggle-tooth cat /etc/passwd >/tmp/passwd.snag
```

This command will copy the super-user's mail file from the remote host into the remote file /tmp/root.mail. Since *cat*(1) must run as the super-user in order to read /usr/mail/root, we must supply the super-user's login name and password following the **host** argument. Notice that the quoting causes output redirection to occur on the remote host.

```
cheval snaggle-tooth "*(root OpenSesame)" "cat /usr/mail/root >/tmp/root.mail"
```

SEE ALSO

chhost(1), chserver(1M).

NAME

chhost – construct a pathname for connecting to a CHAOSnet host

SYNOPSIS

chhost host

DESCRIPTION

Chhost looks up the *CHAOSnet* host number of the specified host and returns a pathname of the form /dev/chaos/*dddd* where *dddd* is the decimal host number for the remote host. In order to use this path for connecting to the host, the user must append a *slash* ('/'), and a contact name for the desired server running on the remote host.

Chhost is primarily useful within shell scripts, most notably *cheval (1)*. As an example, here is the *cheval* script:

```
#!/bin/sh

Usage="Usage: `basename $0` <host> <cmd> [ args... ]"
case $# in
[01])   echo $Usage; exit 1;;
*)      ;;
esac

host="$1"; shift

cat "`chhost $host`/EVAL $*"
```

SEE ALSO

cheval(1), chserver(1M).

NAME

chmod – change mode

SYNOPSIS

chmod mode files

DESCRIPTION

The permissions of the named *files* are changed according to *mode*, which may be absolute or symbolic. An absolute *mode* is an octal number constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit, 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

A symbolic *mode* has the form:

[*who*] *op permission* [*op permission*]

The *who* part is a combination of the letters **u** (user), **g** (group) and **o** (other). The letter **a** stands for all (**ugo**), the default if *who* is omitted.

Op can be + to add *permission* to the file's mode, – to take away *permission*, or = to assign *permission* absolutely (all other bits will be reset).

Permission is any combination of the letters **r** (read), **w** (write), **x** (execute), **s** (set owner or group ID), and **t** (save text, or sticky); **u**, **g**, or **o** indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with = to take away all permissions.

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter **s** is useful only with **u** or **g**; **t** works only with **u**.

Only the owner of a file (or the superuser) may change its mode.

EXAMPLES

The first example denies write permission to others, the second makes a file executable:

chmod o– w file

chmod +x file

SEE ALSO

ls(1), chmod(2).

NAME

chown, chgrp – change owner or group

SYNOPSIS

chown owner file ...

chgrp group file ...

DESCRIPTION

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

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

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(2), group(4), passwd(4).

NAME

chsend - send message to users

SYNOPSIS

chsend [**- m**] [**user1, user2, user3@ loc1, user4@ loc2...**] [**- f** *filename*]

DESCRIPTION

Chsend reads in a message or a file, and sends it to the named users. The message receivers can be either local or network users, distinguished by a '@', which separates the user's name from the machine he is using. Each receiver that is logged on gets the message printed on his screan. If a receiver is logged on at several terminals, the message is printed at up to three of them. If a local receiver is not logged on when the message is sent, the program asks if it should send mail to him. This does not occur if a network receiver is not logged on.

When the program prompts for the message with "Msg:", just type the message. When you are done, type '^D'. If you are sending a long message, it is better to make a file, and use the *-f* option.

The arguments (user names, file names, and options) can appear in any order. The program prints diagnostics of who received a message and who received mail.

The following options are interpreted by **chsend**.

- m Send mail to all receivers who are not currently logged on.

- f*filename*
Instead of reading in a message, send the file *filename* to the receivers.

FILES

/usr/spool/sends/* spool area

SEE ALSO

wall(1), mail(1)

DIAGNOSTICS

Lists the users to whom a message was sent, and to whom mail was sent.

BUGS

Does not do aliasing.

NAME

chtime – return the time-of-day as maintained on a remote CHAOSnet host

SYNOPSIS

chtime hosts...

DESCRIPTION

Chtime reports the time-of-day according to the remote *hosts*. Output consists of the hosts name, its CHAOSnet address in octal, and a date string in the form of *date*(1).

SEE ALSO

chsettime(1M)

NAME

ck – checkout device status, lock and free devices

SYNOPSIS

ck [-key] [device]

DESCRIPTION

Ck checks out device owners and status. Its actions are controlled by the *key* argument. The *key* is one character preceded by a dash. The other argument to the command specifies which *device* is to be checked out. If no key is specified, the named device is locked.

The key can be one of the following letters:

f The named *device* is freed for use by any processor. This will not work if the device is owned by a processor other than the 68010.

t The named *device* is taken from whichever processor owns it. Unlike **f** this command works even if the device is owned by another processor. This is generally the most useful key.

a All device names and their statuses are listed. No *device* argument is given with this command.

The following devices may be specified:

half-inch-tape
quarter-inch-tape
ttya
ttyb
ethernet

The following device statuses may be shown:

free
not present
idle for *idletime*
owned by *processor*

FILES

/etc/ck
/dev/rmt/0m?
/dev/mt/0m?
/dev/rqt/0m?
/dev/ttya
/dev/ttyb
/etc/devlock
/dev/sysconf

DIAGNOSTICS

Complains if you try to free or lock a device you do not own.
Complains if you specify an unknown device.

BUGS

No known bugs.

NAME

clear – clear terminal screen

SYNOPSIS

clear

DESCRIPTION

Clear clears your screen if this is possible. It looks in the environment for the terminal type and then in */etc/termcap* to figure out how to clear the screen.

FILES

/etc/termcap terminal capability data base

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 codes returned are: 0 for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

col – filter reverse line-feeds

SYNOPSIS

col [– **bfpx**]

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-line-feeds (**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 is 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-line-feeds (**ESC-9**), but never contains either kind of reverse line motion.

Unless the – **x** option is given, *col* converts 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, new-line, SI, SO, VT (\013), and ESC followed by **7**, **8**, or **9**. The VT character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

Normally, *col* ignores any unknown escape sequences found in the 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.

SEE ALSO

nroff(1), tbl(1).

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* is a device that can interpret half-line motions; otherwise, use – **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 [– **o**] [– **s**] [– p*sid*] [– c*list*] files

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) to reconstruct the given SCCS files. The reconstructed files should 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 – is the name 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.

The generated shell procedure is written on the standard output.

Keyletter arguments are as follows. 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 *SCCS ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

– c*list* A *list* (see *get*(1) for the syntax of a *list*) of deltas to be preserved. 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, produces a report giving, for each file: the filename, 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, one should use this option to determine exactly how much space is saved by the combining process.

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

FILES

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

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(4).
"Source Code Control System User's Guide" in the *User's Guide*.

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 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 filename – 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** is a no-op.

SEE ALSO

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

NAME

cp, ln, mv – copy, link or move files

SYNOPSIS

cp file1 [file2 ...] target
ln file1 [file2 ...] target
mv 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 *mv* determines that the mode of *target* forbids writing, it prints the mode (see *chmod*(2)) and reads the standard input for one line (if the standard input is a terminal); if the line begins with **y**, the move takes place; if not, *mv* exits.

Only *mv* allows *file1* to be a directory, in which case the directory rename occurs only if the two directories have the same parent.

SEE ALSO

cpio(1), rm(1), chmod(2).

BUGS

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

Ln does not link across file systems.

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio – o [**acBv**]

cpio – i [**BcdmrtuvfsSb6**] [patterns]

cpio – p [**adlmruv**] 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.

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 given in the name-generating notation of *sh*(1). In *patterns*, metacharacters **?**, *****, and [...] match the slash / character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is ***** (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below.

Cpio – p (pass) reads the standard input to obtain a list of pathnames 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.

B Block input/output 5,120 bytes to the record (does not apply to the *pass* option; meaningful only with data directed to or from **/dev/rmt?**).

d Create *directories* as needed.

c Write *header* information in ASCII character form for portability.

r Interactively *rename* files. If the user types a null line, the file is skipped.

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

u Copy *unconditionally* (normally, an older file cannot replace a newer file with the same name).

v *Verbose*: print a list of filenames. When used with the **t** option, the table of contents looks like the output of an **ls – l** command (see *ls*(1)).

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

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

f Copy in all files except those in *patterns*.

s Swap bytes. Use only with the **– i** option.

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

b Swap both bytes and halfwords. Use only with the **– i** option.

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

EXAMPLES

The first example below copies the contents of a directory into an archive; the second duplicates a directory hierarchy:

```
ls | cpio – o >/dev/mt0

cd olddir
find . – depth – print | cpio – pdl newdir
```

The trivial case **find . – depth – print | cpio – oB >/dev/rmt0** can be handled more efficiently by:

find . – cpio /dev/rmt0

SEE ALSO

ar(1), find(1), cpio(4).

BUGS

Pathnames are restricted to 128 characters. If there are too many unique linked files, the program runs out of memory; thereafter, linking information is lost. Only the superuser can copy special files.

NAME

cpp – the C language preprocessor

SYNOPSIS

/lib/cpp [option ...] [ifile [ofile]]

DESCRIPTION

Cpp is the C language preprocessor which is invoked as the first pass of any C compilation using the *cc*(1) command. The output of *cpp* 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 in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(1) command since the functionality of *cpp* may someday be moved elsewhere. See *m4*(1) for a general macro processor.

Cpp optionally accepts two filenames as arguments. *Ifile* is the input and *ofile* is the 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 Pass along all comments except those found on *cpp* directive lines. By default, *cpp* strips C-style comments.

– U*name*

Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The current list of these possibly reserved symbols includes:

operating system:	ibm, gcos, os, tss, unix
hardware:	interdata, pdp11, u370, u3b, vax, m68k
UNIX System variant:	RES, RT

– D*name*

– D*name*=*def*

Define *name* as if by a **#define** directive. If no =*def* is given, *name* is defined as 1.

– 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. When this option is used, **#include** files whose names are enclosed in "" are searched for first in the directory of the *ifile* 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 *ifile* argument is not searched.

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 filename (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by #. 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 tokens, and a) by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma-separated list.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on.

#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** option above for more detail.

#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 where it comes from. If "*filename*" is not given, the current filename is unchanged.

#endif
Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**.

#ifdef *name*
The lines following 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**.

#ifndef *name*
The lines following do not 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**.

#if *constant-expression*
Lines following 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.

#else Reverses the notion of the test directive that matches this directive. If lines previous to this directive are ignored, the following lines appear in the output. If lines previous to this directive are not ignored, the following lines do not appear in the output.

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

FILES

/usr/include	standard directory for **#include** files

SEE ALSO
cc(1), m4(1).

DIAGNOSTICS
The error messages produced by *cpp* are self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NOTES
When newline characters were found in argument lists for macros to be expanded, previous versions of *cpp* put out the newlines as they were found and expanded. The current version of *cpp* replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

NAME

csh – a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [– **cefinstvVxX**] [arg ...]

DESCRIPTION

Csh is a command language interpreter. It begins by executing commands from the file '.cshrc' in the *home* directory of the invoker. If this is a login shell then it also executes commands from the file '.login' there. In the normal case, the shell will then begin reading commands from the terminal, prompting with '% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the file '.logout' in the users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters '&' '|' ';' '<' '>' '(' ')' form separate words. If doubled in '&&', '||', '<<' or '>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with '\'. A newline preceded by a '\' is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, '`', ''' or '"', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of ''' or '"' characters a newline preceded by a '\' gives a true newline character.

When the shell's input is not a terminal, the character '#' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by '\' and in quotations using '`', ''', and '"'.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ';', and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an '&'. Such a sequence is automatically prevented from being terminated by a hangup signal; the *nohup* command need not be used.

Any of the above may be placed in '(' ')' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with '||' or '&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See *Expressions.)*

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus history substitutions provide a generalization of a *redo* function.

History substitutions begin with the character '!' and may begin **anywhere** in the input stream if a history substitution is not already in progress. This '!' may be preceded by an '\' to prevent its special meaning; a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. History substitutions also occur when an input line begins with '↑'. This special abbreviation will be described later.

Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list, the size of which is controlled by the *history* variable. The previous command is always retained. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the history command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an '!' in the prompt string.

With the current event 13 we can refer to previous events by event number '!11', relatively as in '!-2' (referring to the same event), by a prefix of a command word as in '!d' for event 12 or '!w' for event 9, or by a string contained in a word in the command as in '!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '!!' refers to the previous command; thus '!!' alone is essentially a *redo*. The form '!#' references the current command (the one being typed in). It allows a word to be selected from further left in the line, to avoid retyping a long name, as in '!#:1'.

To select words from an event we can follow the event specification by a ':' and a designator for the desired words. The words of a input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	*n*'th argument
↑	first argument, i.e. '1'
$	last argument
%	word matched by (immediately preceding) ?*s*? search
x-*y*	range of words
-*y*	abbreviates '0-*y*'
*	abbreviates '↑-$', or nothing if only 1 word in event
*x**	abbreviates '*x*-$'
x-	like '*x**' but omitting word '$'

The ':' separating the event specification from the word designator can be omitted if the argument selector begins with a '↑', '$', '*' '-' or '%'. After the optional word designator can be placed a sequence of modifiers, each preceded by a ':'. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing '.xxx' component, leaving the root name.
s/*l*/*r*/	Substitute *l* for *r*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g. 'g&'.

p Print the new command but do not execute it.
q Quote the substituted words, preventing further substitutions.
x Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a 'g' the modification is applied only to the first modifiable word. In any case it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of '/'; a '\' quotes the delimiter into the *l* and *r* strings. The character '&' in the right hand side is replaced by the text from the left. A '\' quotes '&' also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in '!?*s*?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing '?' in a contextual scan.

A history reference may be given without an event specification, e.g. '!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus '!?foo?↑ !$' gives the first and last arguments from the command matching '?foo?'.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a '↑'. This is equivalent to '!:s↑' providing a convenient shorthand for substitutions on the text of the previous line. Thus '↑lb↑lib' fixes the spelling of 'lib' in the previous command. Finally, a history substitution may be surrounded with '{' and '}' if necessary to insulate it from the characters which follow. Thus, after 'ls – ld ~paul' we might do '!{l}a' to do 'ls – ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ' and "

The quotation of strings by '' and '"' can be used to prevent all or some of the remaining substitutions. Strings enclosed in '' are prevented any further interpretation. Strings enclosed in '"' are yet variable and command expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see *Command Substitution* below) does a '"' quoted string yield parts of more than one word; '' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the *alias* and *unalias* commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for 'ls' is 'ls – l' the command 'ls /usr' would map to 'ls – l /usr', the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep !↑ /etc/passwd' then 'lookup bill' would map to 'grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can 'alias print 'pr \!* | lpr'' to make a command which *pr's* its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle which causes command input to be echoed. The setting of this variable results from the **-v** command line option.

Other operations treat variables numerically. The '@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '$' characters. This expansion can be prevented by preceding the '$' with a '\' except within '"'s where it **always** occurs, and within ''s where it **never** occurs. Strings quoted by '`' are interpreted later (see *Command substitution* below) so '$' substitution does not occur there until later, if at all. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in '"' or given the ':q' modifier the results of variable substitution may eventually be command and filename substituted. Within '"' a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the ':q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name
${name}

Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}

May be used to select only some of the words from the value of *name*. The selector is subjected to '$' substitution and may consist of a single number or two numbers separated by a '-'. The first word of a variables value is numbered '1'. If the first number of a range is omitted it defaults to '1'. If the last member of a range is omitted it defaults to '$#name'. The selector '*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}

Gives the number of words in the variable. This is useful for later use in a '[selector]'.

$0

Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}

Equivalent to '$argv[number]'.

$*

Equivalent to '$argv[*]'.

The modifiers ':h', ':t', ':r', ':q' and ':x' may be applied to the substitutions above as may ':gh', ':gt' and ':gr'. If braces '{' '}' appear in the command form then the modifiers must appear within the braces. **The current implementation allows only one ':' modifier on each '$' expansion.**

The following substitutions may not be modified with ':' modifiers.

$?name
${?name}

Substitutes the string '1' if name is set, '0' if it is not.

$?0

Substitutes '1' if the current input filename is know, '0' if it is not.

$$

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

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in '`'. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within '"'s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters '*', '?', '[' or '{' or begins with the character '~', then that word is a candidate for filename substitution, also known as 'globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the meta-characters '*', '?' and '[' imply pattern matching, the characters '~' and '{' being more akin to abbreviations.

In matching filenames, the character '.' at the beginning of a filename or immediately following a '/', as well as the character '/' must be matched explicitly. The character '*' matches any string of characters, including the null string. The character '?' matches any single character. The sequence '[...]' matches any one of the characters enclosed. Within '[...]', a pair of characters separated by '- ' matches any character lexically between the two.

The character '~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. '~' it expands to the invokers home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits and '-' characters the shell searches for a user with that name and substitutes their home directory; thus '~ken' might expand to '/usr/ken' and '~ken/chmach' to '/usr/ken/chmach'. If the character '~' is followed by a character other than a letter or '/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation 'a{b,c,d}e' is a shorthand for 'abe ace ade'. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus '~source/s1/{oldls,ls}.c' expands to '/usr/source/s1/oldls.c /usr/source/s1/ls.c' whether or not these files exist without any chance of error if the home directory for 'source' is '/usr/source'. Similarly '../{memo,*box}' might expand to '../memo ../box ../mbox'. (Note that 'memo' was not sorted with the results of matching '*box'.) As a special case '{', '}' and '{}' are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name
: Open file *name* (which is first variable, command and filename expanded) as the standard input.

<< word
: Read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting '\', '"', '`' or '`' appears in *word* variable and command substitution is performed on the intervening lines, allowing '\' to quote '$', '\' and '`'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name
: The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

  If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g. a terminal or '/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the '!' forms can be used and suppress this check.

  The forms involving '&' route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way as '<' input filenames are.

>> name
>>& name
>>! name
>>&! name
: Uses file *name* as standard output like '>' but places output at the end of the file. If the variable *noclobber* is set, then it is an error for the file not to exist unless one of the '!' forms is given. Otherwise similar to '>'.

If a command is run detached (followed by '&') then the default standard input for the command is the empty file '/dev/null'. Otherwise the command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the

command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The '<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form '|&' rather than just '|.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, *exit*, *if*, and *while* commands. The following operators are available:

|| && | ↑ & == != <= >= < > << >> + - * / % ! ~ ( )

Here the precedence increases to the right, '==' and '!=', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '-', '*' '/' and '%' being, in groups, at the same level. The '==' and '!=' operators compare their arguments as strings, all others operate on numbers. Strings which begin with '0' are considered octal numbers. Null or missing arguments are considered '0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in '{' and '}' and file enquiries of the form '-*l* name' where *l* is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. '0'. Command executions succeed, returning true, i.e. '1', if the command exits with status 0, otherwise they fail, returning false, i.e. '0'. If more detailed status information is required then the command should be executed outside of an expression and the variable *status* examined.

Control flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if*-*then*-*else* form of the *if* statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias name
alias name wordlist
The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified *wordlist* as the alias of *name; wordlist* is command and filename substituted. *Name* is not allowed to be *alias* or *unalias*

alloc
Shows the amount of dynamic core in use, broken down into used and free core, and address of the last location in the heap. With an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

break
Causes execution to resume after the *end* of the nearest enclosing *forall* or *while*. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a *switch*, resuming after the *endsw*.

case label:
A label in a *switch* statement as discussed below.

cd
cd name
chdir
chdir name
Change the shells working directory to directory *name*. If no argument is given then change to the home directory of the user.

If *name* is not found as a subdirectory of the current directory (and does not begin with '/', './', or '../'), then each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with '/', then this is tried to see if it is a directory.

continue
Continue execution of the nearest enclosing *while* or *foreach*. The rest of the commands on the current line are executed.

default:
Labels the default case in a *switch* statement. The default should come after all *case* labels.

echo wordlist
The specified words are written to the shells standard output. A '\c' causes the echo to complete without printing a newline, akin to the '\c' in *nroff*(1). A '\n' in wordlist causes a newline to be printed. Otherwise the words are echoed, separated by spaces.

else
end
endif
endsw
See the description of the *foreach, if, switch,* and *while* statements below.

exec command
The specified command is executed in place of the current shell.

exit
exit(expr)
The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

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

The builtin command *continue* may be used to continue the loop prematurely and the builtin command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with '?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob wordlist
Like *echo* but no '\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word
The specified *word* is filename and command expanded to yield a string of the form 'label'. The shell rewinds its input as much as possible and searches for a line of the form 'label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

history
Displays the history event list.

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

if (expr) **then**
...
else if (expr2) **then**
...
else
...
endif
If the specified *expr* is true then the commands to the first *else* are executed; else if *expr2* is true then the commands to the second else are executed, etc. Any number of *else-if* pairs are possible; only one *endif* is needed. The *else* part is likewise optional. (The words *else* and *endif* must appear at the beginning of input lines; the *if* must appear alone on its input line or after an *else.)*

login
Terminate a login shell, replacing it with an instance of **/bin/login.** This is one way to log off, included for compatibility with **/bin/sh.**

logout
Terminate a login shell. Especially useful if *ignoreeof* is set.

nice
nice + number
nice command
nice + number command

The first form sets the *nice* for this shell to 4. The second form sets the *nice* to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user may specify negative niceness by using 'nice - number ...'. Command is always executed in a sub-shell, and the restrictions place on commands in simple *if* statements apply.

nohup
nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. On the Computer Center systems at UC Berkeley, this also *submits* the process. Unless the shell is running detached, *nohup* has no effect. All processes detached with "&" are automatically *nohup'ed.* (Thus, *nohup* is not really needed.)

onintr
onintr -
onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form 'onintr - ' causes all interrupts to be ignored. The final form causes the shell to execute a 'goto label' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

rehash

Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command

The specified *command* which is subject to the same restrictions as the *command* in the one line *if* statement above, is executed *count* times. I/O redirections occurs exactly once, even if *count* is 0.

set
set name
set name=word
set name[index]=word
set name=(wordlist)

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word.* The fourth form sets the *index'th* component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist.* In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv name value

(Version 7 systems only.) Sets the value of environment variable *name* to be *value*, a single string. Useful environment variables are 'TERM' the type of your terminal and 'SHELL' the shell you are using.

shift

shift variable

The members of *argv* are shifted to the left, discarding *argv[1]*. It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name

The shell reads commands from *name*. *Source* commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Input during *source* commands is **never** placed on the history list.

switch (string)
case str1:
...
breaksw
...
default:
...
breaksw
endsw

Each case label is successively matched, against the specified *string* which is first command and filename expanded. The file metacharacters '*', '?' and '[...]' may be used in the case labels, which are variable expanded. If none of the labels match before a 'default' label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* causes execution to continue after the *endsw*. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the *endsw*.

time

time command

With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the *time* variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask

umask value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by 'unalias *'. It is not an error for nothing to be *unaliased*.

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables

are removed by 'unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be *unset*.

wait

All child processes are waited for. It the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (expr)

...

end

While the specified expression evaluates non-zero, the commands between the *while* and the matching end are evaluated. *Break* and *continue* may be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the *foreach* statement if the input is a terminal.

@

@ name = expr

@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains '<', '>', '&' or '|' then at least this part of the expression must be placed within '(' ')'. The third form assigns the value of *expr* to the *index'th* argument of *name*. Both *name* and its *index'th* component must already exist.

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

Special postfix '++' and '--' operators increment and decrement *name* respectively, i.e. '@ i++'.

Pre-defined variables

The following variables have special meaning to the shell. Of these, *argv, child, home, path, prompt, shell* and *status* are always set by the shell. Except for *child* and *status* this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

The shell copies the environment variable PATH into the variable *path*, and copies the value back into the environment whenever *path* is set. Thus is is not necessary to worry about its setting other than in the file *.cshrc* as inferior *csh* processes will import the definition of *path* from the environment. (It could be set once in the *.login* except that commands through *net*(1) would not see the definition.)

argv Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. '$1' is replaced by '$argv[1]', etc.

cdpath Gives a list of alternate directories searched to find subdirectories in *chdir* commands.

child The process number printed when the last command was forked with '&'. This variable is *unset* when this process terminates.

echo Set when the **-x** command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

histchars Can be assigned a two character string. The first character is used as a history character in place of "!", the second character is used in place of the "^" substitution mechanism. For example, "set histchars=",;"" will cause the history characters to be comma and semicolon.

history Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.

home The home directory of the invoker, initialized from the environment. The filename expansion of '~' refers to this variable.

ignoreeof If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.

mail The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says 'You have new mail.' if the file exists with an access time not greater than its modify time.

If the first word of the value of *mail* is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says 'New mail in *name*' when there is mail in the file *name*.

noclobber As described in the section on *Input/output*, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that '>>' redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. 'echo [' still gives an error.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable then only full path names will execute. The usual search path is '.', '/bin' and '/usr/bin', but this may vary from system to system. For the super-user the default search path is '/etc', '/bin' and '/usr/bin'. A shell which is given neither the **-c** nor the **-t** option will normally hash the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.

prompt The string which is printed before each command is read from an interactive terminal input. If a '!' appears in the string it will be replaced by the current event number unless a preceding '\' is given. Default is '% ', or '# ' for the super-user.

shell The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of *Non-builtin Command Execution* below.) Initialized to the (system-dependent) home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status '1', all other builtin commands set status '0'.

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the **-v** command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via *exec*(2). Each word in the variable *path* names a directory from which the shell will attempt to execute the command. If it is given neither a **-c** nor a **-t** option, the shell will hash the names in these directories into an internal table so that it will only try an *exec* in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via *unhash*), or if the shell was given a **-c** or **-t** argument, and in any case for each directory component of *path* which does not begin with a "/", the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus '(cd ; pwd) ; pwd' prints the *home* directory; leaving you where you were (printing this after the home directory), while 'cd ; pwd' leaves you in the *home* directory. Parenthesized commands are most often used to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands an a new shell is spawned to read it.

If there is an *alias* for *shell* then the words of the alias will be prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (e.g. '$shell'). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

Argument list processing

If argument 0 to the shell is '-' then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.

-e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell will start faster, because it will neither search for nor execute commands from the file '.cshrc' in the invokers home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A '\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.

-V Causes the *verbose* variable to be set even before '.cshrc' is executed.

-X Is to **-x** as **-V** is to **-v**.

After processing of flag arguments if arguments remain but none of the **-c, -i, -s,** or **-t** options was given the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a 'standard' shell if the first character of a script is not a '#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable *argv*.

Signal handling

The shell normally ignores *quit* signals. The *interrupt* and *quit* signals are ignored for an invoked command if the command is followed by '&'; otherwise the signals have the values which the shell inherited from its parent. The shells handling of interrupts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file '.logout'.

AUTHOR

William Joy

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after '.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Standard shell, for shell scripts not starting with a '#'.
/tmp/sh*	Temporary file for '<<'.
/dev/null	Source of empty file.
/etc/passwd	Source of home directories for '~name'.

LIMITATIONS

Words can be no longer than 512 characters. The number of characters in an argument varies from system to system. Early version 6 systems typically have 512 character limits while later version 6 and version 7 systems have 5120 character limits. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Also command substitutions may substitute no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

SEE ALSO

access(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), a.out(5), environ(5), 'An introduction to the C shell'

BUGS

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

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

It should be possible to use the ':' modifiers on the output of command substitutions. All and more than one ':' modifier should be allowed on '$' substitutions.

Some commands should not touch *status* or it may be so transient as to be almost useless. Oring in 0200 to *status* on abnormal termination is a kludge.

In order to be able to recover from failing *exec* commands on version 6 systems, the new command inherits several open files other than the normal standard input and output and diagnostic output. If the input and output are redirected and the new command does not close these files, some files may be held open unnecessarily.

There are a number of bugs associated with the importing/exporting of the PATH. For example, directories in the path using the ~ syntax are not expanded in the PATH. Unusual paths, such as (), can cause csh to core dump.

This version of *csh* does not support or use the process control features of the 4th Berkeley Distribution. It contains a number of known bugs which have been fixed in the process control version. This version is not supported.

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*.

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 user's responsibility to remove it.

EXAMPLES

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

This example command creates four files, **cobol00 ... cobol03**. After the split files have been edited, 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 splits 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 is still 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 creates a file containing each separate C routine (up to 21) in **prog.c**.

SEE ALSO

ed(1), sh(1), regexp(5).

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

ct – spawn getty to a remote terminal

SYNOPSIS

ct [**–h**] [**–v**] [**–wn**] [**–s**speed] telno ...

DESCRIPTION

Ct dials the phone number of a modem that is attached to a terminal, and spawns a *getty* process to that terminal. *Telno* is a telephone number, with equal signs for secondary dial tones and minus signs for delays at appropriate places. If more than one telephone number is specified, *ct* tries each in succession until one answers; this is useful for specifying alternate dialing paths.

Ct tries each line listed in the file */usr/lib/uucp/L-devices* until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, *ct* asks if it should wait for one, and if so, for how many minutes it should wait before it gives up. *Ct* continues 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.

Normally, *ct* hangs up the current line, so that that line can answer the incoming call. The **–h** option prevents this action. 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 300.

After the user on the destination terminal logs out, *ct* prompts, **Reconnect?** . If the response begins with the letter **n**, the line is dropped; otherwise, *getty* is started again and the **login:** prompt is printed.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

FILES

/usr/lib/uucp/L-devices
/usr/adm/ctlog

SEE ALSO

cu(1C), login(1), uucp(1C).

NAME

ctags – create a tags file

SYNOPSIS

ctags [– **BFatuwvx**] name ...

DESCRIPTION

Ctags makes a tags file for *ex*(1) from the specified C, Pascal and Fortran sources. A tags file gives the locations of specified objects (in this case functions and typedefs) in a group of files. Each line of the tags file contains the object name, the file in which it is defined, and an address specification for the object definition. Functions are searched with a pattern, typedefs with a line number. Specifiers are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these objects definitions.

If the – **x** flag is given, *ctags* produces a list of object 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. This is a simple index which can be printed out as an off-line readable function index.

If the – **v** flag is given, an index of the form expected by *vgrind*(1) is produced on the standard output. This listing contains the function name, file name, and page number (assuming 64 line pages). Since the output will be sorted into lexicographic order, it may be desired to run the output through **sort – f**. Sample use:

```
ctags – v files |sort – f > index
vgrind – x index
```

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. Others are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are processed again looking for C definitions.

Other options are:

– **F** use forward searching patterns (/.../) (default).

– **B** use backward searching patterns (?...?).

– **a** append to tags file.

– **t** create tags for typedefs.

– **w** suppressing warning diagnostics.

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

The tag *main* is treated specially in C programs. The tag formed is created by prepending *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.

FILES

tags	output tags file

SEE ALSO

ex(1), vi(1)

AUTHOR

Ken Arnold; FORTRAN added by Jim Kleckner; Bill Joy added Pascal and – **x**, replacing *cxref;* C typedefs added by Ed Pelegri-Llopart.

BUGS

Recognition of **functions**, **subroutines** and **procedures** for FORTRAN and Pascal is done is a very simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name you lose.

The method of deciding whether to look for C or Pascal and FORTRAN functions is a hack.

Does not know about #ifdefs.

Should know about Pascal types. Relies on the input being well formed to detect typedefs. Use of -tx shows only the last line of typedefs.

NAME

cu – call another system

SYNOPSIS

cu [– sspeed] [– lline] [– **h**] [– **t**] [– **d**] [– **m**] [– **o**|– **e**] telno | **dir**

DESCRIPTION

Cu calls up another system, a terminal, or possibly a non- system. It manages an interactive conversation with possible transfers of ASCII files. *Speed* gives the transmission speed (110, 150, 300, 600, 1200, 4800, 9600); 300 is the default value. Most modems are either 300 or 1200 baud. For dial-out lines, *cu* chooses a modem speed (300 or 1200) as the slowest available which can handle the specified transmission speed. Directly connected lines may be set to speeds higher than 1200 baud.

The – **l** value may be used to specify a device name for the communications line device to be used. This can be used to override searching for the first available line having the right speed. The speed of a line is taken from the file */usr/lib/uucp/L-devices*, overriding any speed specified by the – **s** option. The – **h** option emulates local echo, supporting calls to other computer systems which expect terminals to be in half-duplex mode. The – **t** option is used when dialing an ASCII terminal which has been set to auto-answer. Appropriate mapping of carriage-returns to carriage-return-line-feed pairs is set. The – **d** option causes diagnostic traces to be printed. The – **m** option specifies a direct line which has modem control. The – **e** (– **o**) option designates that even (odd) parity is to be generated for data sent to the remote. *Telno* is the telephone number, with equal signs for secondary dial tone or minus signs for delays, at appropriate places. The string **dir** for *telno* may be used for directly connected lines, and implies a null ACU. Using **dir** insures that a line has been specified by the – **l** option.

Cu tries each line listed in the file */usr/lib/uucp/L-devices* until it finds an available line with appropriate attributes or runs out of entries. 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:

~.	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.
~%**take** *from* [*to*]	Copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~%**put** *from* [*to*]	Copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places.
~~...	Send the line ~... to the remote system.
~%**nostop**	Turn off the DC3/DC1 input control protocol for the remainder of the session. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters,

The *receive* process normally copies data from the remote system to its standard output. A line from the remote that begins with ~> initiates an output diversion to a file. The complete sequence is:

~>[>]: *file*
zero or more lines to be written to *file*
~>

Data from the remote is diverted (or appended, if >> is used) to file. The trailing ~> terminates 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 ones 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, **stty tabs** mode should be set on the remote system if tabs are to be copied without expansion.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..(tty-device)
/dev/null

SEE ALSO

cat(1), ct(1C), echo(1), stty(1), uucp(1C).

DIAGNOSTICS

Exit code is zero for normal exit, non-zero (various values) otherwise.

BUGS

Cu buffers input internally.
There is an artificial slowing of transmission by *cu* during the **~%put** operation so that loss of data is unlikely.

NAME

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

SYNOPSIS

cut – clist [file1 file2 ...]
cut – flist [– **d**char] [– **s**] [file1 file2 ...]

DESCRIPTION

Use *cut* to remove columns from a table or fields from each line of a file; in data base parlance, *cut* implements the projection of a relation. The fields 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 can be marked with a field delimiter character such as *tab* (– **f** option). *Cut* can be used as a filter; if no files are given, the standard input is used.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional – to indicate ranges as in the – **o** option of *nroff/troff* for page 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** (no space) 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 are passed through intact (useful for table subheadings), unless – **s** is specified.

– **d***char* The character following – **d** (no space) 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 are passed through untouched.

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

HINTS

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

The command

cut – d: – f1,5 /etc/passwd

maps user IDs to names.

The command

name=`who am i | cut – f1 – d ' '`

sets **name** to the current login name.

SEE ALSO

grep(1), paste(1).

DIAGNOSTICS

line too long A line can have no more than 511 characters or fields.

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.

no fields The *list* is empty.

NAME

cw, checkcw – prepare constant-width text for troff

SYNOPSIS

cw [**-l**xx] [**-r**xx] [**-f**n] [**-t**] [**+t**] [**-d**] [files]

checkcw [**-l**xx] [**-r**xx] files

DESCRIPTION

Cw is a preprocessor for *troff*(1) input files that contain text to be typeset in the constant-width (CW) font.

Text typeset with the CW font resembles the output of terminals and line printers. This font is used to typeset examples of programs and computer output in user manuals, programming texts, etc. (An earlier version of this font was used in typesetting *The C Programming Language* by B. W. Kernighan and D. M. Ritchie.) It has been designed to be quite distinctive (but not overly obtrusive) when used together with the Times Roman font.

Because the CW font contains a non-standard set of characters and because text typeset with it requires different character and inter-word spacing than is used for standard fonts, documents that use the CW font must be preprocessed by *cw*.

The CW font contains the 94 printing ASCII characters:

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!$%&()‘’*+-.,/:;=?[]|-_ fiflffffi°

plus 8 non-ASCII characters represented by 4-character *troff*(1) names (in some cases attaching these names to non-standard graphics):

Character	*Symbol*	*Troff Name*
“Cents” sign	¢	@c **t**
EBCDIC “not” sign	¬	@n **o**
Left arrow	←	@f **i-**
Right arrow	→	@- **fl**
Down arrow	↓	@d **a**
Vertical single quote	’	@f **m**
Control-shift indicator	▵	@d **g**
Visible space indicator	†	@s **q**
Hyphen	-	@h **y**

The hyphen is a synonym for the unadorned minus sign (-). Certain versions of *cw* recognize two additional names: @u **a** for an up arrow and @l **h** for a diagonal left-up (home) arrow.

Cw recognizes 5 request lines, as well as user-defined delimiters. The request lines look like *troff*(1) macro requests, and are copied in their entirety by *cw* onto its output; thus, they can be defined by the user as *troff*(1) macros; in fact, the **.CW** and **.CN** macros should be so defined (see *HINTS* below). The 5 requests are:

.CW Start of text to be set in the CW font; **.CW** causes a break; it can take precisely the same options, in precisely the same format, as are available on the *cw* command line.

.CN End of text to be set in the CW font; **.CN** causes a break; it can take the same options as are available on the *cw* command line.

.CD Change delimiters and/or settings of other options; takes the same options as are available on the *cw* command line.

.CP *arg1 arg2 arg3 . . . argn*
All the arguments (which are delimited like *troff*(1) macro arguments) are

concatenated, with the odd-numbered arguments set in the CW font and the even-numbered ones in the prevailing font.

.PC *arg1 arg2 arg3 ... argn*
Same as **.CP**, except that the even-numbered arguments are set in the CW font and the odd-numbered ones in the prevailing font.

The **.CW** and **.CN** requests are meant to bracket text (e.g., a program fragment) that is to be typeset in the CW font "as is." Normally, *cw* operates in the *transparent* mode. In that mode, except for the **.CD** request and the nine special 4-character names listed in the table above, every character between **.CW** and **.CN** request lines stands for itself. In particular, *cw* arranges for periods (**.**) and apostrophes (**'**) at the beginning of lines, and backslashes (****) everywhere to be "hidden" from *troff*(1). The transparent mode can be turned off (see below), in which case normal *troff*(1) rules apply; in particular, lines that begin with **.** and **'** are passed through untouched (except if they contain delimiters– see below). In either case, *cw* hides the effect of the font changes generated by the **.CW** and **.CN** requests; *cw* also defeats all ligatures (**fi**, **ff**, etc.) in the CW font.

The only purpose of the **.CD** request is to allow the changing of various options other than just at the beginning of a document.

The user can also define *delimiters*. The left and right delimiters perform the same function as the **.CW**/**.CN** requests; they are meant, however, to enclose CW "words" or "phrases" in running text (see example under *BUGS* below). *Cw* treats text between delimiters in the same manner as text enclosed by **.CW**/**.CN** pairs, except that, for aesthetic reasons, spaces and backspaces inside **.CW**/**.CN** pairs have the same width as other CW characters, while spaces and backspaces between delimiters are half as wide, so they have the same width as spaces in the prevailing text (but are *not* adjustable). Font changes due to delimiters are *not* hidden.

Delimiters have no special meaning inside **.CW**/**.CN** pairs.

The options are:

-l *xx* The 1- or 2-character string *xx* becomes the left delimiter; if *xx* is omitted, the left delimiter becomes undefined, which it is initially.

-r *xx* Same for the right delimiter. The left and right delimiters may (but need not) be different.

-f *n* The CW font is mounted in font position *n*; acceptable values for *n* are 1, 2, and 3 (default is 3, replacing the bold font). This option is only useful at the beginning of a document.

-t Turn transparent mode *off*.

+t Turn transparent mode *on* (this is the initial default).

-d Print current option settings on file descriptor 2 in the form of *troff*(1) comment lines. This option is meant for debugging.

Cw reads the standard input when no *files* are specified (or when **-** is specified as the last argument), so it can be used as a filter. Typical usage is:

cw *files* | **troff ...**

Checkcw checks that left and right delimiters, as well as the **.CW**/**.CN** pairs, are properly balanced. It prints out all offending lines.

HINTS

Typical definitions of the **.CW** and **.CN** macros meant to be used with the *mm*(5) macro package:

```
.de CW
.DS I
.ps 9
.vs 10.5p
.ta 16m/3u 32m/3u 48m/3u 64m/3u 80m/3u 96m/3u ...
..
.de CN
.ta 0.5i 1i 1.5i 2i 2.5i 3i 3.5i 4i 4.5i 5i 5.5i 6i
.vs
.ps
.DE
..
```

At the very least, the **.CW** macro should invoke the *troff*(1) no-fill (**.nf**) mode.

When set in running text, the CW font is meant to be set in the same point size as the rest of the text. In displayed matter, on the other hand, it can often be profitably set one point *smaller* than the prevailing point size (the displayed definitions of **.CW** and **.CN** above are one point smaller than the running text on this page). The CW font is sized so that, when it is set in 9-point, there are 12 characters per inch.

Documents that contain CW text may also contain tables and/or equations. If this is the case, the order of preprocessing should be: *cw*, *tbl*, and *eqn*. Usually, the tables contained in such documents will not contain any CW text, although it is entirely possible to have *elements* of the table set in the CW font; of course, care must be taken that *tbl*(1) format information not be modified by *cw*. Attempts to set equations in the CW font are not likely to be either pleasing or successful.

In the CW font, overstriking is most easily accomplished with backspaces: letting ©represent a backspace, **d©©©©dg** yields **d**. Because backspaces are half as wide between delimiters as inside **.CW**/**.CN** pairs, two backspaces are required for each overstrike between delimiters (see paragraph describing delimiters above).

FILES

/usr/lib/font/ftCW CW font-width table

SEE ALSO

eqn(1), mmt(1), tbl(1), troff(1), mm(5), mv(5).

WARNINGS

If text preprocessed by *cw* is to make any sense, it must be set on a typesetter equipped with the CW font or on a STARE facility; on the latter, the CW font appears as bold, but with the proper CW spacing.

Do not use periods (.), backslashes (©), or double quotes ( ) as delimiters, or as arguments to **.CP** and **.PC**.

BUGS

Certain CW characters don't concatenate gracefully with certain Times Roman characters, e.g., a CW ampersand (*&*) followed by a Times Roman comma(,); in such cases, judicious use of *troff*(1) half- and quarter-spaces (©) and ©) is most salutary, e.g., one should use _&_©, (rather than just plain _&_,) to obtain &, (assuming that _ is used for both delimiters).
Use of *cw* with *nroff* is unproductive.
The output of *cw* is hard to read.
See also *BUGS* under *troff*(1).

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

DESCRIPTION

Cxref analyzes a collection of C files and attempts to build a cross-reference table. *Cxref* utilizes a special version of *cpp* to include **#define** 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 identical to their interpretation by *cc*(1)), the following *options* are interpreted by *cxref*:

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

– w<num>
Format output no wider than <num> (decimal) columns. This option defaults to 80 if <num> is not specified or is less than 51.

– o*file* Direct output to named *file*.

– s Operate silently; input filenames not printed.

– t Format listing for 80-column width.

FILES

/usr/lib/xcpp special version of C-preprocessor.

SEE ALSO

cc(1).

DIAGNOSTICS

Error messages are cryptic, but usually mean that you can't compile these files.

NAME

date – print and set the date

SYNOPSIS

date [mmddhhmm[yy]] [+format]

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. The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24-hour system). The second *mm* is the minute number. *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. 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.

If the argument begins with +, the output of *date* is under the control of the user. The format for the output is similar to that of the first argument to *printf*(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by % and is 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 new-line character.

Field Descriptors:

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

EXAMPLE

The command

date '+DATE: %n/%d/%y%nTIME: %H:%M:%S'

generates as output:

DATE: 08/01/76
TIME: 14:45:05

DIAGNOSTICS

No permission you aren't the superuser and are trying to change the date;
bad conversion the date set is syntactically incorrect;
bad format character the field descriptor is not recognizable.

FILES

/dev/kmem

WARNING

It is a bad practice to change the date while the system is running multi-user.

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. 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. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x The top element of the stack is treated as a character string and is executed as a string of *dc* commands.

X The number on the top of the stack is replaced with its scale factor.

[...] The bracketed ASCII string is put 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 The top element on the stack is replaced by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! The rest of the line is interpreted as a command.

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 The output base is pushed 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 The number on the top of the stack is replaced with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : are used by *bc* for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+ dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1), a preprocessor for *dc*, that provides infix notation and a C-like syntax to implement functions and reasonable control structures for programs.

"Interactive Desk Calculator (DC)" in the *Support Tools Guide*.

DIAGNOSTICS

x is unimplemented *x* is an octal number.

stack empty There are not enough elements on the stack to do what was asked.

Out of space The free list is exhausted (too many digits).

Out of headers Too many numbers are being kept.

Out of pushdown Too many items are on the stack.

Nesting Depth There are too many levels of nested execution.

NAME

dd - convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

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

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

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

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case, *cbs* characters are placed into the conversion buffer, converted to ASCII. Trailing blanks are trimmed and a newline is added before sending the line to the output. In the latter case, ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks are added to make up an output record of size *cbs*.

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

EXAMPLE

This command reads an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file **x**:

dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase

Note the use of raw magtape. *Dd* is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

SEE ALSO

cp(1).

DIAGNOSTICS

f+p records in(out) numbers of full and partial records read(written)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256-character standard in the CACM Nov, 1968. The *ibm* conversion, while less accepted as a standard, corresponds better to certain IBM print train conventions. There is no universal solution.

New-lines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

NAME

delta – make a delta (change) to an SCCS file

SYNOPSIS

delta [– rSID] [– s] [– n] [– glist] [– 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 path name 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 it is necessary but 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* Specifies 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 Modification Request (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 new-line 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) for validating 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 new-line 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

All files of the form *?*-file are explained in the "Source Code Control System User's Guide" section of the *User's Guide*. The naming convention for these files is also described there.

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*(5)) and causes 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), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccsfile(4).
"Source Code Control System User's Guide" in the *User's Guide*.

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 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).

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 can be used to 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, but incomplete, job. It works only when changed stretches are short and well-separated; however, it does work on files of unlimited length. Options - **e** and - **f** are unavailable with - **h**.

FILES

/tmp/d?????
/usr/lib/diffh for the - **h** option

SEE ALSO

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 (.).

NAME

diff – differential file and directory comparator

SYNOPSIS

diff [**– l**] [**– r**] [**– s**] [**– cefh**] [**– b**] dir1 dir2
diff [**– cefh**] [**– b**] file1 file2
diff [**– D***string*] [**– b**] file1 file2

DESCRIPTION

If both arguments are directories, *diff* sorts the contents of the directories by name, and then runs the regular file *diff* algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed. Options when comparing directories are:

– l long output format; each text file *diff* is piped through *pr*(1) to paginate it, other differences are remembered and summarized after all text file differences are reported.

– r causes application of *diff* recursively to common subdirectories encountered.

– s causes *diff* to report files which are the same, which are otherwise not mentioned.

– S*name*
starts a directory *diff* in the middle beginning with file *name*.

When run on regular files, and when comparing text files which differ during directory comparison, *diff* tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, *diff* finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, then either may be given as '– ', in which case the standard input is used. If *file1* is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* is used (and vice versa).

There are several options for output format; the default output format 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 '>'.

Except for **– b**, which may be given with any of the others, the following options are mutually exclusive:

– e producing a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. 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
```

Extra commands are added to the output when comparing directories with **– e**, so that the result is a *sh*(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2*.

– f produces a script similar to that of **– e,** not useful with *ed,* and in the opposite order.

– c produces a diff with lines of context. The default is to present 3 lines of context and may be changed, e.g to 10, by **– c10**. With **– c** the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with '– '; those added to *file2* are marked '+ '. Lines which are changed from one file to the other are marked in both files with '!'.

– 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.

– Dstring
causes *diff* to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1,* while defining *string* will yield *file2.*

– b causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

FILES

/tmp/d?????
/usr/lib/diffh for **– h**
/bin/pr

SEE ALSO

cmp(1), cc(1), comm(1), ed(1), diff3(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

BUGS

Editing scripts produced under the **– e** or **– f** option are naive about creating lines consisting of a single '.'.

When comparing directories with the **– b** option specified, *diff* first compares the files ala *cmp,* and then decides to run the *diff* algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical because the only differences are insignificant blank string differences.

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 follow immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file are suppressed.

Under the – **e** option, *diff3* publishes a script for the editor *ed* that incorporates 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 can be used to 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 . negate the effect of option – **e**.
Diff3 cannot process files longer than 64K bytes.

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**] 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 filenames 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.

SEE ALSO

cmp(1), diff(1).

NAME

dis – disassembler

SYNOPSIS

dis [– o] [– V] [– L] [– **d** sec] [– **da** sec] [– **F** function] [– **t** sec] [– **l** string] files

DESCRIPTION

The *dis* command produces an assembly language listing of each of its object *file* arguments. The listing includes assembly statements and 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. Default is hexadecimal.

– **V** Write the version number of the disassembler to standard error.

– **L** Invoke a lookup of C source labels in the symbol table for subsequent printing.

– **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.

– **t** *sec* Disassemble the named section as text.

– **l** *string* Disassemble the library file specified as *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 **/lib**.

If the – **d**, – **da**, or – **t** options are specified, only those named sections from each user supplied filename are disassembled. Otherwise, all sections containing text are disassembled.

If the – **F** option is specified, only those named functions from each user supplied filename are disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as [**5**], represents that the C breakpointable line number starts with the following instruction. An expression such as <**40**> in the operand field, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A C function name will appear in the first column, followed by **( )**.

SEE ALSO

as(1), *cc*(1), *ld*(1).

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

NAME

du – summarize disk usage

SYNOPSIS

du [– **ars**] [names]

DESCRIPTION

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

The optional argument – **s** causes only the grand total for each of the specified *names* to be given. The optional argument – **a** causes an entry to be generated for each file. Absence of both options causes an entry to be generated for each directory only.

Du is normally silent about directories that cannot be read, files that cannot be opened, etc. The – **r** option causes *du* to generate messages in such instances.

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

BUGS

If the – **a** option is not used, non-directories given as arguments are not listed.
If there are too many distinct linked files, *du* counts the excess files more than once.
Files with holes in them get an incorrect block count.

NAME

dump – dump selected parts of an object file

SYNOPSIS

dump [– **acfghlorst**] [– **z** name] files

DESCRIPTION

The *dump* command dumps selected parts of each of its object *file* arguments.

This command accepts 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.

– **f** Dump each file header.

– **g** Dump the global symbols in the symbol table of a 6.0 archive.

– **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.

– *zname* Dump line number entries for the named function.

– **c** Dump the string table.

The following *modifiers* are used in conjunction with the options listed above to modify their capabilities.

– **d***number* Dump the section number or range of sections starting at *number* and ending either at the last section number or *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. When the – **t** is used in conjunction with + **t**, it 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 the – **s** and – **o** options of *dump*.

– *zname,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 on the standard output, separated by blanks and terminated by a new-line. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b	backspace
\c	print line without new-line
\f	form-feed
\n	new-line
\r	carriage return
\t	tab
****	backslash
*****n*	the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a zero.

Echo is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1).

NAME

ed, red – text editor

SYNOPSIS

ed [–] [**– x**] [file]

red [–] [**– x**] [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; i.e., the file is read into *ed*'s buffer so that it can be edited. The optional – 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*. If **– x** is present, an *x* command is simulated first to handle an encrypted file. *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 only allows editing of files in the current directory. It prohibits executing shell commands via *!shell command*. Attempts to bypass these restrictions result in an error message (**restricted shell**).

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

<:t5,10,15 s72:>

tab stops are set at columns 5, 10 and 15, and a maximum line length of 72 is imposed. NOTE: While inputting text, typed tab characters are expanded to every eighth column, as is the default.

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 the command. The addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can 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. Input mode is left by typing a period (.) alone at the beginning of a line.

Ed supports a limited form of regular expression (RE) 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 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 REs match a *single* character:

1.1 An ordinary character (not one of those discussed in 1.2 below) 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 below).

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 ([])

NAME

crypt – encode/decode

SYNOPSIS

crypt [password]

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 *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

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

will print the clear.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* 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 cleartext 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 lower-case letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. To minimize this possibility, *crypt* takes care to destroy any record of the key immediately upon entry. No doubt the choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(8)

BUGS

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, Bell Telephone Laboratories assumes no responsibility for their use by the recipient. Further, Bell Laboratories assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

(see 1.4 below).

c. $ (currency symbol), which is special at the end of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see below how slash (/) is used in the *g* command.)

1.3 A period (.) is a one-character RE that matches any character except new-line.

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches any single character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character except new-line and the remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (-) may be used to indicate a range of consecutive 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 above stand for themselves within such a string of characters.

The following rules may be used to construct REs 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 currency symbol ($) at the end of an entire RE constrains that RE to match a final segment of a line.

3.3 The construction ^*entire RE*$ constrains the entire RE to match the entire line.

3.4 The null RE (e.g., //) is equivalent to the last RE encountered. See also the last paragraph before *FILES* below.

To understand addressing in *ed* it is necessary to know what the current line is at any given time. Generally speaking, the current line is the last line affected by a command; the exact

effect on the current line is discussed under the description of each command. Addresses are constructed as follows:

1. The character . addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. *'x* addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A RE enclosed by slashes (/) addresses the first line containing a matching RE found by searching forward from the line following the current line. 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 *FILES* below.
6. A RE enclosed in question marks (**?**) addresses the first line containing a matching RE found by searching backward from the line preceding the current line. 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 *FILES* below.
7. An address followed by a plus sign (+) or minus sign (−) and a decimal number specifies that address plus (or 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. above). 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 change 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.

(.,.)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, and then the named file to be read in; . is set to the last line of the buffer. If no filename is given, the currently-remembered filename, if any, is used (see the *f* command). The number of characters read is typed; *file* is remembered for possible use as a default filename in subsequent *e*, *r*, and *w* commands. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. Such a shell command is not remembered as the current filename. See also *DIAGNOSTICS* below.

E *file*

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

f *file*

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

(1,$)g/*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* below.

(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 new-line acts as a null command; an & causes the re-execution of the most recent command executed within the current invocation of *G*. Note that the commands input as part of the execution of the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

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

recent ? diagnostic.

H

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

(.)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 new-line characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be a lower-case letter. 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 mnemonic overstrikes; all other non-printing characters are printed in octal and long lines are folded. The *l* command may be appended to any other command except *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 prompts with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is off initially.

q

The *q*uit command causes *ed* to exit. No automatic write of a file is done (but see *DIAGNOSTICS* below).

Q

The editor exits without checking for changes 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 filename is

given, the currently-remembered filename, if any, is used (see *e* and *f* commands). The currently-remembered filename is not changed unless *file* is the very first filename mentioned since *ed* was invoked. Address 0 is legal for *r* and causes the file to be read 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 filename.

(.,.)s/*RE*/*replacement*/ or
(.,.)s/*RE*/*replacement*/**g**

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. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or new-line 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* below.

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 new-line character into it. The new-line in the *replacement* must be escaped by preceding it by \. Such substitution cannot be done as part of a *g* or *v* command list.

(.,.)t*a*

This command acts just like the *m* command, except that a *copy* of the addressed lines is placed after address *a* (which may be 0); . 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 (i.e., 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 *sh*(1)) dictates otherwise. The currently-remembered filename is *not* changed unless *file* is the very first filename mentioned since *ed* was invoked. If no

filename is given, the currently-remembered filename, if any, is used (see *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 filename.

x

A key string is demanded from the standard input. Subsequent *e*, *r*, and *w* commands encrypt and decrypt the text with this key by the algorithm of *crypt*(1). An explicitly empty key turns off encryption.

(**$**)**=**

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

(**.+1**)<new-line>

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

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

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per filename, and 128K characters in the buffer. 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 and all characters after the last new-line. Files (e.g., **a.out**) that contain characters not in the ASCII set (bit 8 on) cannot be edited by *ed*.

If the closing delimiter of a RE or of a replacement string (e.g., /) would be the last character before a new-line, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

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

FILES

/tmp/e#	temporary; # is the process number.
ed.hup	work is saved here if the terminal is hung up.

SEE ALSO

crypt(1), grep(1), sed(1), sh(1), stty(1), fspec(4), regexp(5).
"A Tutorial Introduction to the *UNIX Text Editor*" by B. W. Kernighan.
"Advanced Editing on *UNIX*" by B. W. Kernighan.
"Tutorial - Text Editor" in the *User's Guide*.
"Document Preparation" in the *Document Processing Guide*.

DIAGNOSTICS

?	command errors
? *file*	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 destroys the buffer. The – command-line option inhibits this feature.

WARNINGS AND 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 the editor is invoked from a restricted shell (see *sh*(1)).
The sequence **\n** in a RE does not match a new-line character.
The *l* command mishandles DEL.
Files encrypted directly with the *crypt*(1) command with the null key cannot be edited.
Characters are masked to 7 bits on input.

NAME

edit – text editor (variant of ex for casual users)

SYNOPSIS

edit [**– r**] 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. The following brief introduction should help you get started with *edit*. A more complete basic introduction is provided by tutorial materials in the *User's Guide*. See *ex*(1) for other useful documents; in particular, if you are using a CRT terminal you may want to learn about the display editor *vi*.

BRIEF INTRODUCTION

To edit the contents of an existing file, begin with the following command to the shell:

edit *filename*

Edit makes a copy of the file which you can then edit, and tells you how many lines and characters are in the file. To create a new file, just make up a name for the file and try to run *edit* on it; you will cause an error diagnostic, but don't worry.

Edit prompts for commands with the character ':', which you should see after starting the editor. If you are editing an existing file, then you have some lines in the *edit* buffer (its name for the copy of the file you are editing). Most commands to *edit* use the "current line" if you don't specify which line to use. Thus, if you type **print** (which can be abbreviated **p**) and hit carriage return (as you should after all *edit* commands), the current line is printed. If you **delete** (**d**) the current line, *edit* prints the new current line. When you start editing, *edit* makes the last line of the file the current line. If you **delete** this last line, then the new last line becomes the current one. In general, after a **delete,** the next line in the file becomes the current line. (Deleting the last line is a special case.)

If you start with an empty file, or wish to add some new lines, then the **append** (**a**) command can be used. After you give this command (typing a carriage return after the word **append**) *edit* reads lines from your terminal, placing these lines after the current line. You terminate the input process by typing a line consisting of just a ".". The last line of text before the "." line becomes the current line. The command **insert** (**i**) is like **append** but places the lines you give before, rather than after, the current line.

Edit numbers the lines in the buffer, the first line having number 1. If you give the command "**1**" then *edit* types this first line. If you then give the command **delete**, *edit* deletes the first line, line 2 becomes line 1, and *edit* prints the current line (the new line 1) so you can see where you are. In general, the current line is always 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. You type "s/*old*/*new*/", where *old* is replaced by the characters you want to eliminate and *new* is the new characters you want to insert.

The command **file** (**f**) tells you how many lines are in the buffer you are editing and says "[Modified]" if you have changed it. After modifying a file you can put the buffer text back to replace the file by giving a **write** (**w**) command. You can then leave the editor by issuing a **quit** (**q**) command. If you run *edit* on a file, but don't 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 are warned that there has been "No **write** since last change" and *edit* awaits another command. If you wish not to **write** the buffer out then you can issue another **quit** command. The buffer is then irretrievably discarded, and you return to the shell.

By using the **delete** and **append** commands, and giving line numbers to see lines in the file, you can make any changes you desire. You should learn at least a few more things, however, if you are to use *edit* more than a few times.

The **change** (**c**) command changes the current line to a sequence of lines you supply (as with **append** you terminate **change** with a line consisting of only a "."). You can tell **change** to change more than one line by giving the line numbers of the lines you want to change, i.e., "3,5change". You can print lines this way too. Thus "1,23p" prints the first 23 lines of the file.

The **undo** (**u**) command reverses the effect of the last command that changed the buffer. Thus, if a **substitute** command doesn't do what you want, you can type **undo** and the old contents of the line are restored. You can also **undo** an **undo** command so that you can continue to change your mind. *Edit* issues a warning message when commands you give affect more than one line of the buffer. If the amount of change seems unreasonable, type *undo* and look to see what happened. If you decide that the change is ok, then you can type *undo* again to get it back. Note that commands such as *write* and *quit* cannot be undone.

To look at the next line in the buffer, hit carriage return. To look at a number of lines, hit ^D (control key and, while it is held down, D key, then let up both) rather than carriage return. This shows you a half screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at the text surrounding the current line by giving the command "z.". The current line becomes the last line printed; you can get back to the line where you were before the "z." command by saying "'". The **z** command can also be given following characters. **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 you can specify the number you want. For example, **z.12** produces 12 lines of text. This method of giving counts can be used with other commands. You can delete 5 lines starting with the current line with the command **delete 5 .**

You can use line numbers to find things in a file; since the line numbers change when you insert and delete lines, this is somewhat unreliable. You can search backward and forward 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 the text, it wraps, end around, and continues to search back to the current line. A useful feature 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 a symbolic name "."; this is most useful in a range of lines, as in **.,$print**, which prints the rest of the lines in the file. To get to the last line in the file you can refer to it by its symbolic name "$". Thus the command **$ delete** or **$d** deletes the last line in the file, no matter which line was the current line before. Arithmetic with line references is also possible. Thus the line **$– 5** is the fifth before the last, and **.+20** is 20 lines after the present one.

You can find out the line number of the current line by typing **.=**. This is useful if you wish to move or copy a section of text within a file or between files. Find out the first and last line numbers you wish to copy or move (say 10 to 20). For a move you can then type **10,20delete a**, which deletes these lines from the file and places them in a buffer named *a*. *Edit* has 26 such buffers named *a* through *z*. You can later get these lines back by typing **put a** to put the contents of buffer *a* after the current line. If you want to move or copy these lines between files you can give an **edit** (**e**) command after copying the lines, following it with the name of the other file you wish to edit, i.e., **edit chapter2**. By changing *delete* to *yank* in the command shown above, you can get a pattern for copying lines. If the text you wish to move or copy is all within one file it is not necessary to use named buffers. **10,20move $**, for example, moves lines 10 through 20 to the end of the file.

SEE ALSO

ex (1), vi (1),
"Edit: A tutorial", by Ricki Blau and James Joyce.
"Text Editors" in the *User's Guide*.

BUGS

See *ex*(1).

NAME

efl – Extended Fortran Language

SYNOPSIS

efl [options] [files]

DESCRIPTION

Efl compiles a program written in the EFL language into clean Fortran on the standard output. *Efl* provides the C-like control constructs of *ratfor*(1):

statement grouping with braces.

decision-making:

if, **if-else**, and **select-case** (also known as **switch-case**);
while, **for**, Fortran **do**, **repeat**, and **repeat** ... **until** loops;
multi-level **break** and **next**.

EFL has C-like data structures, e.g.:

```
struct
        {
        integer flags(3)
        character(8) name
        long real coords(2)
        } table(100)
```

The language offers generic functions, assignment operators (+=, &=, etc.), and sequentially evaluated logical operators (&& and ||). There is a uniform input/output syntax:

```
write(6,x,y:f(7,2), do i=1,10 { a(i,j),z.b(i) })
```

EFL also provides some syntactic "sugar":

free-form input:
multiple statements per line; automatic continuation; statement label names (not just numbers).

comments:
this is a comment.

translation of relational and logical operators:
>, >=, &, etc., become .GT., .GE., .AND., etc.

return expression to caller from function:
return (*expression*)

defines:
define *name replacement*

includes:
include *file*

Efl understands several option arguments: – **w** suppresses warning messages, – # suppresses comments in the generated program, and the default option – **C** causes comments to be included in the generated program.

An argument with an embedded = (equal sign) sets an EFL option as if it had appeared in an **option** statement at the start of the program. Many options are described in the reference manual cited below. A set of defaults for a particular target machine may be selected by one of the choices: **system=unix**, **system=gcos**, or **system=cray**. The default setting of the **system** option is the same as the machine the compiler is running on. Other specific options determine the style of input/output, error handling, continuation conventions, the number of characters packed per word, and default formats.

Efl is best used with *f77*(1).

SEE ALSO

cc(1), f77(1), ratfor(1).

The Programming Language EFL by S.I. Feldman.

NAME

enable, disable – enable/disable LP printers

SYNOPSIS

enable printers
disable [– **c**] [– **r**[reason]] printers

DESCRIPTION

Enable activates the named *printers*, enabling them to print requests taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

Disable deactivates the named *printers*, preventing them from printing requests taken by *lp*(1). By default, any requests that are currently printing on the designated printers are 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 useful with *disable* are:

– **c** Cancel any requests that are currently printing on any of the designated printers.

– **r**[*reason*] Associates a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next – **r** option. If the – **r** option is not present or the – **r** option is given without a reason, then a default reason is used. *Reason* is reported by *lpstat*(1).

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).

NAME

eqn, neqn, checkeq - format mathematical text for nroff or troff

SYNOPSIS

eqn [- dxy] [- pn] [- sn] [- fn] [files]

neqn [- dxy] [- pn] [- sn] [- fn] [files]

checkeq [files]

DESCRIPTION

Eqn is a *troff*(1) preprocessor for typesetting mathematical text on a phototypesetter, while *neqn* is used for the same purpose with *nroff* on typewriter-like terminals. Usage is almost always:

```
eqn files | troff
neqn files | nroff
```

or equivalent.

If no files are specified (or if - is specified as the last argument), these programs read the standard input. A line beginning with .EQ marks the start of an equation; the end of an equation is marked by a line beginning with .EN. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to designate two characters as *delimiters*; subsequent text between delimiters is then treated as *eqn* input. Delimiters may be set to characters *x* and *y* with the command-line argument - d*xy* or (more commonly) with **delim** *xy* between .EQ and .EN. The left and right delimiters may be the same character; the dollar sign is often used as such a delimiter. Delimiters are turned off by **delim off**. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

The program *checkeq* reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within *eqn* are separated by spaces, tabs, new-lines, braces, double quotes, tildes, and circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character such as *x* could appear, a complicated construction enclosed in braces may be used instead. Tilde (~) represents a full space in the output, circumflex (^) half as much.

Subscripts and superscripts are produced with the keywords **sub** and **sup**. Thus *x sub j* makes x_j, *a sub k sup 2* produces a_k^2, while $e^{x^2+y^2}$ is made with *e sup {x sup 2 + y sup 2}*. Fractions are made with **over**: *a over b* yields $\frac{a}{b}$; **sqrt** makes square roots: *1 over sqrt {ax sup 2+ bx+ c}* results in $\frac{1}{\sqrt{ax^2+bx+c}}$.

The keywords **from** and **to** introduce lower and upper limits: $\lim_{n\to\infty}\sum_{0}^{n} x_i$ is made with *lim from {n - > inf } sum from 0 to n x sub i*. Left and right brackets, braces, etc., of the right height are made with **left** and **right**: *left [x sup 2 + y sup 2 over alpha right] ~=~ 1* produces $\left[x^2+\frac{y^2}{\alpha}\right]=1$. Legal characters after **left** and **right** are braces, brackets, bars, **c** and **f** for ceiling and floor, and "" for nothing at all (useful for a right-side-only bracket). A **left** *thing* need not have a matching **right** *thing*.

Vertical piles of things are made with **pile**, **lpile**, **cpile**, and **rpile**: *pile {a above b above c}* produces $\begin{matrix} a \\ b \\ c \end{matrix}$. Piles may have arbitrary numbers of elements; **lpile** left-justifies, **pile** and **cpile** center (but with different vertical spacing), and **rpile** right justifies. Matrices are made with **matrix**: *matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }* produces $\begin{matrix} x_i & 1 \\ y_2 & 2 \end{matrix}$. In addition, there is **rcol** for a right-justified column.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **tilde**, **bar**, **vec**, **dyad**, and **under**: *x dot = f(t) bar* is $\dot{x} = \overline{f(t)}$, *y dotdot bar ~=~ n under* is $\ddot{\overline{y}} = \underline{n}$, and *x vec ~=~ y dyad* is $\vec{x} = \vec{\vec{y}}$.

Point sizes and fonts can be changed with **size** *n* or **size** ± *n*, **roman**, **italic**, **bold**, and **font** *n*. Point sizes and fonts can be changed globally in a document by **gsize** *n* and **gfont** *n*, or by the command-line arguments – **s***n* and – **f***n*.

Normally, subscripts and superscripts are reduced by 3 points from the previous size; this may be changed by the command-line argument – **p***n*.

Successive display arguments can be lined up. Place **mark** before the desired lineup point in the first equation; place **lineup** at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with **define**:

define thing % replacement %

defines a new token called *thing* that will be replaced by *replacement* whenever it appears thereafter. The *%* may be any character that does not occur in *replacement*.

Keywords such as **sum** ($\sum$), **int** ($\int$), **inf** (∞), and shorthands such as >= ($\geq$), != ($\neq$), and – > ($\rightarrow$) are recognized. Greek letters are spelled out in the desired case, as in **alpha** (α), or GAMMA (Γ). Mathematical words such as **sin**, **cos**, and **log** are made Roman automatically. *Troff*(1) four-character escapes such as \(**dd** (‡) and \(**bs** (⊘) may be used anywhere. Strings enclosed in double quotes ("...") are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *troff*(1) when all else fails. Full details are given in the manual cited below.

SEE ALSO

cw(1), mm(1), mmt(1), nroff(1), tbl(1), troff(1), eqnchar(5), mm(5), mv(5).
"Formatting Facilities (Mathematics Typesetting Program" in the *Document Processing Guide*.
"Typesetting Mathematics– User's Guide" by B. W. Kernighan and L. L. Cherry.

BUGS

To embolden digits, parentheses, etc., it is necessary to quote them, as in **bold "12.3"**.
See also *BUGS* under *troff*(1).

NAME

error – analyze and disperse compiler error messages

SYNOPSIS

error [**– n**] [**– s**] [**– q**] [**– v**] [**– t** suffixlist] [**– I** ignorefile] [name]

DESCRIPTION

Error analyzes and optionally disperses the diagnostic error messages produced by a number of compilers and language processors to the source file and line where the errors occurred. It can replace the painful, traditional methods of scribbling abbreviations of errors on paper, and permits error messages and source code to be viewed simultaneously without machinations of multiple windows in a screen editor.

Error looks at the error messages, either from the specified file *name* or from the standard input, and attempts to determine which language processor produced each error message, determines the source file and line number to which the error message refers, determines if the error message is to be ignored or not, and inserts the (possibly slightly modified) error message into the source file as a comment on the line preceding to which the line the error message refers. Error messages which can't be categorized by language processor or content are not inserted into any file, but are sent to the standard output. *Error* touches source files only after all input has been read. By specifying the **– q** query option, the user is asked to confirm any potentially dangerous (such as touching a file) or verbose action. Otherwise *error* proceeds on its merry business. If the **– t** touch option and associated suffix list is given, *error* will restrict itself to touch only those files with suffices in the suffix list. Error also can be asked (by specifying **– v**) to invoke *vi*(1) on the files in which error messages were inserted; this obviates the need to remember the names of the files with errors.

Error is intended to be run with its standard input connected via a pipe to the error message source. Some language processors put error messages on their standard error file; others put their messages on the standard output. Hence, both error sources should be piped together into *error*. For example, when using the *csh* syntax,

make – s lint |& error – q – v

will analyze all the error messages produced by whatever programs *make* runs when making lint.

Error knows about the error messages produced by: *make, cc, cpp, ccom, as, ld, lint, pi, pc* and *f77*. *Error* knows a standard format for error messages produced by the language processors, so is sensitive to changes in these formats. For all languages except *Pascal,* error messages are restricted to be on one line. Some error messages refer to more than one line in more than one files; *error* will duplicate the error message and insert it at all of the places referenced.

Error will do one of six things with error messages.

synchronize

Some language processors produce short errors describing which file it is processing. *Error* uses these to determine the file name for languages that don't include the file name in each error message. These synchronization messages are consumed entirely by *error*.

discard Error messages from *lint* that refer to one of the two *lint* libraries, */usr/lib/llib-lc* and */usr/lib/llib-port* are discarded, to prevent accidently touching these libraries. Again, these error messages are consumed entirely by *error*.

nullify Error messages from *lint* can be nullified if they refer to a specific function, which is known to generate diagnostics which are not interesting. Nullified error messages are not inserted into the source file, but are written to the standard output. The names of functions to ignore are taken from either the file named *.errorrc* in the

users's home directory, or from the file named by the **-I** option. If the file does not exist, no error messages are nullified. If the file does exist, there must be one function name per line.

not file specific
Error messages that can't be intuited are grouped together, and written to the standard output before any files are touched. They will not be inserted into any source file.

file specific Error message that refer to a specific file, but to no specific line, are written to the standard output when that file is touched.

true errors Error messages that can be intuited are candidates for insertion into the file to which they refer.

Only true error messages are candidates for inserting into the file they refer to. Other error messages are consumed entirely by *error* or are written to the standard output. *Error* inserts the error messages into the source file on the line preceding the line the language processor found in error. Each error message is turned into a one line comment for the language, and is internally flagged with the string "###" at the beginning of the error, and "%%%" at the end of the error. This makes pattern searching for errors easier with an editor, and allows the messages to be easily removed. In addition, each error message contains the source line number for the line the message refers to. A reasonably formatted source program can be recompiled with the error messages still in it, without having the error messages themselves cause future errors. For poorly formatted source programs in free format languages, such as C or Pascal, it is possible to insert a comment into another comment, which can wreak havoc with a future compilation. To avoid this, programs with comments and source on the same line should be formatted so that language statements appear before comments.

Options available with *error* are:

-n Do *not* touch any files; all error messages are sent to the standard output.

-q The user is *queried* whether s/he wants to touch the file. A "y" or "n" to the question is necessary to continue. Absence of the **-q** option implies that all referenced files (except those referring to discarded error messages) are to be touched.

-v After all files have been touched, overlay the visual editor *vi* with it set up to edit all files touched, and positioned in the first touched file at the first error. If *vi* can't be found, try *ex* or *ed* from standard places.

-t Take the following argument as a suffix list. Files whose suffixes do not appear in the suffix list are not touched. The suffix list is dot separated, and "*" wildcards work. Thus the suffix list:

".c.y.foo*.h"

allows *error* to touch files ending with ".c", ".y", ".foo*" and ".y".

-s Print out *statistics* regarding the error categorization. Not too useful.

Error catches interrupt and terminate signals, and if in the insertion phase, will orderly terminate what it is doing.

AUTHOR

Robert Henry

FILES

~/.errorrc	function names to ignore for *lint* error messages
/dev/tty	user's teletype

BUGS

Opens the teletype directly to do user querying.

Source files with links make a new copy of the file with only one link to it.

Changing a language processor's format of error messages may cause *error* to not understand the error message.

Error, since it is purely mechanical, will not filter out subsequent errors caused by 'floodgating' initiated by one syntactically trivial error. Humans are still much better at discarding these related errors.

Pascal error messages belong after the lines affected (error puts them before). The alignment of the '|' marking the point of error is also disturbed by *error.*

Error was designed for work on CRT's at reasonably high speed. It is less pleasant on slow speed terminals, and has never been used on hardcopy terminals.

NAME

ex – text editor

SYNOPSIS

ex [–] [– **v**] [– **t** *tag*] [– **r**] [+*command*] [– **l**] [– **x**] name ...

DESCRIPTION

Ex is the root of a family of editors which includes *edit, ex* and *vi. Ex* is a line oriented editor which is a superset of *ed.*

If you have a CRT terminal, you may wish to use the display based editor *vi* (see *vi*(1)), which focuses on the display editing portion of *ex.*

FOR ED USERS

If you have used *ed,* you will find that *ex* has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi. Ex* uses many more terminal capabilities than *ed* does. It uses the data base *termcap*(5) and your terminal type (from the variable TERM in the environment) 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 is the central mode of editing when using *vi*(1)). There is also an interline editing command, **open** (**o**), which works on all terminals.

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives access to windows of text. Hitting ^D causes the editor to scroll a half-window of text, which is useful for quickly stepping through a file. Of course, the screen oriented **visual** mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. *Ex* gives you feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command. This makes it easy to detect when a command has affected more lines than you intended.

The editor normally prevents overwriting existing files so that you can't accidentally clobber a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the **recover** command to retrieve your work. This gets 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 edit each in turn. The **next** command can also be given a list of filenames, or a pattern (as used by the shell) to specify a new set of files to be edited. In general, filenames in the editor may be formed with full shell metasyntax. The metacharacter '*%*' is also available in forming filenames and is replaced by the name of the current file. For editing large groups of related files you can use *ex's* **tag** command to quickly locate functions and other important points in any of the files. This is useful when working on a large program when you want to quickly find the definition of a particular function. The command *ctags*(1) builds a *tags* file or a group of C programs.

For moving text between files and within a file the editor has a group of buffers, named *a* through *z.* You can place text in these named buffers and carry it over when you edit another file.

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 very useful option is *autoindent*, which allows the editor to automatically supply leading white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new features include an intelligent **join** (**j**) command which 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*.

INVOCATION OPTIONS

The following invocation options are interpreted by *ex*:

–	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*	Recover *file* after an editor or system crash. If *file* is not specified a list of all saved files is printed.
+ *command*	Begin editing by executing the specified editor search or positioning *command*.
– **l**	**LISP** mode; indents appropriately for lisp code. The () {} [[and]] commands in *vi* and *open* are modified to have meaning for *lisp* .
– **x**	Encryption mode; a key is prompted for allowing creation or editing of an encrypted file.

The *name* argument indicates files to be edited.

Ex States

Command	Normal and initial state. Input prompted for by **:**. Your kill character cancels partial command.
Insert	Entered by **a**, **i**, and **c**. Arbitrary text may be entered. Insert is terminated normally by a line having only a period (**.**) on it, or abnormally with an interrupt.
Open/visual	Entered by **open** or **vi**; terminated with **Q** or ^\.

Ex command names and abbreviations

abbrev	**ab**	next	**n**	unabbrev	**una**
append	**a**	number	**nu**	undo	**u**
args	**ar**	open	**o**	unmap	**unm**
change	**c**	preserve	**pre**	version	**ve**
copy	**co**	print	**p**	visual	**vi**
delete	**d**	put	**pu**	write	**w**
edit	**e**	quit	**q**	xit	**x**
file	**f**	read	**re**	yank	**ya**
global	**g**	recover	**rec**	*window*	**z**
insert	**i**	rewind	**rew**	*escape*	**!**
join	**j**	set	**se**	*lshift*	<
list	**l**	shell	**sh**	*print next*	CR
map		source	**so**	*resubst*	&
mark	**ma**	stop	**st**	*rshift*	>
move	**m**	substitute	**s**	*scroll*	^D

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	environmental variable for options
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Useful options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp		( ) { } are s-exp's
list		print ^I for tab, $ at end
magic		. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start ...
redraw		simulate smart terminal
scroll		command mode lines
sections	sect	macro names ...
shiftwidth	sw	for < >, and input ^D
showmatch	sm	to) and } as typed
slowopen	slow	stop updates during insert
window		visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning pattern formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*
[↑*str*]	... not in *str*
[*x*- *y*]	... between *x* and *y*
*	any number of preceding

FILES

/usr/lib/ex?.?strings	error messages
/usr/lib/ex?.?recover	recover command
/usr/lib/ex?.?preserve	preserve command
/etc/termcap	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 preservation directory

SEE ALSO

awk(1), ed(1), grep(1), vi(1), termcap(5).
"Ex Reference Manual" in the *User's Guide*.
Document Processing Guide.
"An Introduction to Display Editing with Vi" in the *User's Guide*.
"A Tutorial Introduction to the UNIX Text Editor" by Brian W. Kernighan.
"Advanced Editing on UNIX" by Brian W. Kernighan.
"VI/EX Quick Reference Card", University of California, Berkeley.

WARNINGS AND BUGS

The *undo* command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The *z* command prints a 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 don't print a name if the command line '– ' 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

expand, unexpand – expand tabs to spaces, and vice versa

SYNOPSIS

expand [– tabstop] [– tab1,tab2,...,tabn] [file ...]
unexpand [– **a**] [file ...]

DESCRIPTION

Expand processes the named files or the standard input writing the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. *Expand* is useful for pre-processing character files (before sorting, looking at specific columns, etc.) that contain tabs.

If a single *tabstop* argument is given then tabs are set *tabstop* spaces apart instead of the default 8. If multiple tabstops are given then the tabs are set at those specific columns.

Unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default only leading blanks and tabs are reconverted to maximal strings of tabs. If the – **a** option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

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, 2's 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 path name (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).

EXIT CODE

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.

DIAGNOSTICS

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

f77 – Fortran 77 compiler

SYNOPSIS

f77 [options] files

DESCRIPTION

F77 is the

Fortran 77 compiler; it accepts several types of *file* arguments:

Arguments whose names end with **.f** are taken to be Fortran 77 source programs; they are compiled and each object program is left in the current directory in a file whose name is that of the source, with **.o** substituted for **.f**.

Arguments whose names end with **.r** or **.e** are taken to be RATFOR or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by *f77*, producing **.o** files.

In the same way, arguments whose names end with **.c** or **.s** are taken to be C or assembly source programs and are compiled or assembled, producing **.o** files.

The following *options* have the same meaning as in *cc*(1) (see *ld*(1) for link editor options):

– **c**	Suppress link editing and produce **.o** files for each source file.
– **p**	Prepare object files for profiling (see *prof*(1)).
– **O**	Invoke an object code optimizer.
– **S**	Compile the named programs and leave the assembler language output in corresponding files whose names are suffixed with **.s**. (No **.o** files are created.)
– **o***output*	Name the final output file *output*, instead of **a.out**.
– **f**	In systems without floating-point hardware, use a version of *f77* that handles floating-point constants and links the object program with the floating-point interpreter.
– **g**	Generate additional information needed for the use of *sdb*(1)

The following *options* are peculiar to *f77*:

– **onetrip**	Compile DO loops that are performed at least once if reached. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)
– **1**	Same as – **onetrip**.
– **66**	Suppress extensions which enhance Fortran 66 compatibility.
– **C**	Generate code for run-time subscript range-checking.
– **I**[**24s**]	Change the default size of integer variables (only valid on machines where the "normal" integer size is not equal to the size of a single precision real). – **I2** causes all integers to be 2-byte quantities, – **I4** (default) causes all integers to be 4-byte quantities, and – **Is** changes the default size of subscript expressions (only) from the size of an integer to 2 bytes.
– **U**	Do not "fold" cases. *F77* is normally a no-case language (i.e. **a** is equal to **A**). The – **U** option causes *f77* to treat upper and lower cases separately.
– **u**	Make the default type of a variable *undefined*, rather than using the default Fortran rules.
– **w**	Suppress all warning messages. If the option is – **w66**, only Fortran 66 compatibility warnings are suppressed.
– **F**	Apply EFL and RATFOR preprocessor to relevant files and put the result in files whose names have their suffix changed to **.of**. (No **.o** files are created.)
– **m**	Apply the M4 preprocessor to each EFL or RATFOR source file before transforming with the *ratfor*(1) or *efl*(1) processors.

-E The remaining characters in the argument are used as an EFL flag argument whenever processing a **.e** file.

-R The remaining characters in the argument are used as a RATFOR flag argument whenever processing a **.r** file.

Other arguments are taken to be link editor option arguments, *f77*-compilable object programs (typically produced by an earlier run), or libraries of *f77*-compilable routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the default name **a.out**.

FILES

file.[fresc]	input file
file.o	object file
a.out	linked output
./fort[*pid*].?	temporary
/usr/lib/f77pass1	compiler
/lib/f1	pass 2
/lib/c2	optional optimizer
/usr/lib/libF77.a	intrinsic function library
/usr/lib/libI77.a	Fortran I/O library
/lib/libc.a	C library; see Section 3 of this Manual.

SEE ALSO

A Portable Fortran 77 Compiler by S. I. Feldman and P. J. Weinberger.
asa(1), cc(1), efl(1), fsplit(1), ld(1), m4(1), prof(1), ratfor(1), sdb(1).

DIAGNOSTICS

The diagnostics produced by *f77* itself are self-explanatory. Occasional messages may be produced by the link editor *ld*(1).

NAME

factor - factor a number

SYNOPSIS

factor [number]

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2×10^{16}) it factors the number and prints its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If *factor* is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime.

DIAGNOSTICS

Ouch input out of range or garbage input.

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* prints the version stamp, provided it is greater than 0 (see *ld*(1)).

If the **– f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

File uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of */etc/magic* explains its format.

The **– m** option instructs *file* to use an alternate magic file.

The **– c** flag 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**.

FILES

/etc/magic

SEE ALSO

ld(1).

NAME

find – find files

SYNOPSIS

find pathname-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each pathname in the *pathname-list* (i.e., one or more pathnames) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, –*n* means less than *n*, and *n* means exactly *n*.

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

– **perm** *onum* — True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(2)) become significant and the flags are compared:

(flags&onum)===onum

– **type** *c* — True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, fifo (a.k.a named pipe), or plain file.

– **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* — True if the file is *n* blocks long (512 bytes per block).

– **atime** *n* — True if the file has not been accessed in *n* days.

– **mtime** *n* — True if the file has not been modified in *n* days.

– **ctime** *n* — True if the file has not 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* — Write the current file on *device* in *cpio*(4) format (5120 byte records).

– **newer** *file* — True if the current file has been modified more recently than the argument *file*.

(*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

cpio(1), sh(1), test(1), stat(2), cpio(4), fs(4).

NAME

finger – user information lookup program

SYNOPSIS

finger [options] name ...

DESCRIPTION

By default *finger* lists the login name, full name, terminal name and write status (as a '*' before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current UNIX user. (Idle time is minutes if it is a single integer, hours and minutes if a ':' is present, or days and hours if a 'd' is present.)

A longer format also exists and is used by *finger* whenever a list of peoples names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file *.plan* in their home directory, and the project on which they are working from the file *.project* also in the home directory.

Finger options include:

- **– m** Match arguments only on user name.
- **– l** Force long output format.
- **– p** Suppress printing of the *.plan* files
- **– s** Force short output format.

FILES

/etc/utmp	who file
/etc/passwd	for users names, offices, ...
/usr/adm/lastlog	last login times
~/.plan	plans
~/.project	projects

SEE ALSO

w(1), who(1)

AUTHOR

Earl T. Cohen

BUGS

Only the first line of the *.project* file is printed.

The encoding of the gcos field is UCB dependent – it knows that an office '197MC' is '197M Cory Hall', and that '529BE' is '529B Evans Hall'.

A user information data base is in the works and will radically alter the way the information that *finger* uses is stored. Finger will require extensive modification when this is implemented.

NAME

fmt – simple text formatter

SYNOPSIS

fmt [name ...]

DESCRIPTION

Fmt is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces on standard output a version of its input with lines as close to 72 characters long as possible. The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing.

Fmt is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the *ex* editor (e.g. *vi*) the command

```
!}fmt
```

will reformat a paragraph, evening the lines.

SEE ALSO

nroff(1), mail(1)

AUTHOR

Kurt Shoens

BUGS

The program was designed to be simple and fast – for more complex operations, the standard text processors are likely to be more appropriate.

NAME

fold – fold long lines for finite width output device

SYNOPSIS

fold [– width] [file ...]

DESCRIPTION

Fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using *expand*(1) before coming to *fold*.

SEE ALSO

expand(1)

BUGS

If underlining is present it may be messed up by folding.

NAME

fsplit – split f77, ratfor, or efl files

SYNOPSIS

fsplit options files

DESCRIPTION

Fsplit splits the named *file(s)* into separate files, with one procedure per file. A procedure includes *blockdata*, *function*, *main*, *program*, and *subroutine* program segments. Procedure *X* is put in file *X*.**f**, *X*.**r**, or *X*.**e** depending on the language option chosen, with the following exceptions: *main* is put in the file *MAIN*.[**efr**] and unnamed *blockdata* segments in the files *blockdataN*.[**efr**] where *N* is a unique integer value for each file.

The following *options* pertain:

– **f** (default) Input files are *f77*.

– **r** Input files are *ratfor*.

– **e** Input files are *Efl*.

– **s** Strip *f77* input lines to 72 or fewer characters with trailing blanks removed.

SEE ALSO

csplit(1), efl(1), f77(1), ratfor(1), split(1).

NAME

get – get a version of an SCCS file

SYNOPSIS

get [– **r**SID] [– **c**cutoff] [– **i**list] [– **x**list] [– **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 filename by simply removing the leading **s.**; (see also *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 apply 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 below 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 which 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; that is, – **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 one 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

– **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 *gets* 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*.

– 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. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

– x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the – **i** keyletter for the *list* format.

– **k** Suppresses replacement of identification keywords (see below) 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 which 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 ("top") delta in a given release (e.g., – **r1**), or release and level (e.g., – **r1.2**).

– 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, and users should not use it. If both the – **r** and – **a** keyletters are specified, the – **a** keyletter is used. Care should be taken when using the – **a** keyletter in conjunction with the – **e** keyletter, as the SID of the delta to be created may not be what one 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 filename is printed (preceded by a new-line) 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.

IDENTIFICATION 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 (see *admin*(1)).
%F%	SCCS filename.
%P%	Fully qualified SCCS filename.
%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 shouldn't 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 the program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non- program files. %A% = %Z%%Y% %M% %I%%Z%

FILES

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 filename is formed from the SCCS filename: the last component of all SCCS filenames 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, for file **s.xyz.c**, the auxiliary filenames 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 or not 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;
   * otherwise.
b. A blank character if the delta was applied or wasn't applied and ignored;
   * if the delta wasn't applied and wasn't 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 new-line. 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.

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1), sccsfile(4).
"Source Code Control System" in the *Support Tools Guide* and the *User's Guide*.

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 doesn't, then only one file may be named when the – **e** keyletter is used.

NAME

getopt – parse command options

SYNOPSIS

**set – – `getopt** optstring $*`

DESCRIPTION

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* recognizes it; otherwise, *getopt* generates it; in either case, *getopt* places it at the end of the options. The shell's positional parameters ($1 $2 ...) 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 one might 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 accepts 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

sh(1), getopt(3C).

DIAGNOSTICS

Getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

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 Teletypewriter Model 37 terminal (which is the *nroff* default 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)). The following *terminal*s are recognized currently:

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), eqn(1), hp(1), mm(1), tplot(1G), nroff(1), environ(5), greek(5), term(5).

NAME

grep, egrep, fgrep – search a file for a pattern

SYNOPSIS

grep [options] expression [files]

egrep [options] [expression] [files]

fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular *expressions* in the style of *ed*(1); it uses a compact non-deterministic algorithm. *Egrep* patterns are full regular *expressions*; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed *strings*; it is fast and compact. The following *options* are recognized:

- **– v** All lines but those matching are printed.
- **– x** (Exact) only lines matched in their entirety are printed (*fgrep* only).
- **– c** Only a count of matching lines is printed.
- **– l** Only the names of files with matching lines are listed (once), separated by new-lines.
- **– n** Each line is preceded by its relative line number in the file.
- **– b** Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
- **– s** The error messages produced for nonexistent or unreadable files are suppressed (*grep* only).
- **– e** *expression*
  Same as a simple *expression* argument, but useful when the *expression* begins with a – (does not work with *grep*).
- **– f** *file*
  The regular *expression* (*egrep*) or *strings* list (*fgrep*) is taken from the *file*.

In all cases, the filename is output if there is more than one input file. Care should be taken when using the characters $, *, [, ^, |, (,), and \ in *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes '...'.

Fgrep searches for lines that contain one of the *strings* separated by new-lines.

Egrep accepts regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a new-line match strings that are matched by either.
4. A regular expression may be enclosed in parentheses ( ) for grouping.

The order of precedence of operators is [], then *? +, then concatenation, then | and new-line.

SEE ALSO

ed(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*, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.
Lines are limited to 256 characters; longer lines are truncated.

Egrep does not recognize ranges, such as **[a– z]**, in character classes.

NAME

grep – search a file for a pattern

SYNOPSIS

grep [option] ... expression [file] ...

DESCRIPTION

Grep searchs the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular expressions in the style of *ex*(1); it uses a compact nondeterministic algorithm. The following options are recognized.

– v All lines but those matching are printed.

– c Only a count of matching lines is printed.

– l The names of files with matching lines are listed (once) separated by newlines.

– n Each line is preceded by its relative line number in the file.

– b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

– i The case of letters is ignored in making comparisons — that is, upper and lower case are considered identical.

– s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

– w The expression is searched for as a word (as if surrounded by '\<' and '\>', see *ex*(1).)

– e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a – .

In all cases the file name is shown if there is more than one input file. Care should be taken when using the characters $ * [^ |( ) and \ in the *expression* as they are also meaningful to the Shell. It is safest to enclose the entire *expression* argument in single quotes ' '.

In the following description 'character' excludes newline:

A \ followed by a single character other than newline matches that character.

The character ^ matches the beginning of a line.

The character $ matches the end of a line.

A . (period) matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [ ] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in 'a– z0– 9'. A] may occur only as the first character of the string. A literal – must be placed where it can't be mistaken as a range indicator.

A regular expression followed by an * (asterisk) matches a sequence of 0 or more matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

The order of precedence of operators at the same parenthesis level is [ ] then * then concatenation then newline.

SEE ALSO

ex(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.

BUGS

Lines are limited to 256 characters; longer lines are truncated.

NAME

head – give first few lines

SYNOPSIS

head [– count] [file ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1)

NAME

help – ask for help

SYNOPSIS

help [args]

DESCRIPTION

Help finds information to explain a message from a command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, *help* prompts for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1	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., **ge6**, for message 6 from the *get* command).
type 2	Does not contain numerics (as a command, such as **get**)
type 3	Is all numeric (e.g., **212**)

The response of the program is the explanatory information related to the argument, if there is any.

When all else fails, try the command **help stuck**.

FILES

/usr/lib/help	directory containing files of message text.
/usr/lib/help/helploc	file containing locations of help files not in **/usr/lib/help**.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

hostat – check status of Chaosnet hosts

SYNOPSIS

hostat [hosts]

DESCRIPTION

Hostat reports on the status of Chaosnet hosts. Zero or more host names may be supplied on the command line. If no nost names are supplied, all hosts present in the binary host-table **/etc/hostbin** are polled for status.

The following items are reported (numbers are reported in decimal unless indicated otherwise): hostname, octal host address, octal subnet number, number of packets received and transmitted, number of packets aborted and lost, number of packets having invalid checksums, number of packets having an invalid length, and number of packets rejected for lack of system buffer space.

If a host does not respond within a timeout period (about 15 seconds), *hostat* skips it and goes on to the next host. The user may cause a host to be skipped before the timeout period expires by typing the *interrupt* character on the terminal. Two *interrupt* characters typed in rapid succession will abort *hostat* entirely.

NAME

hp – handle special functions of HP 2640 and 2621-series 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. A typical use is:

nroff – h files ... | hp

Regardless of the hardware options on your terminal, *hp* tries to do sensible things with underlining and reverse line-feeds. 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 new-lines. Any contiguous sequence of 3 or more new-lines is converted into a sequence of only 2 new-lines; 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.

For Greek and other special characters, *hp* provides the same set as *300*(1), except that "not" is approximated by a right arrow, and only the top half of the integral sign is shown. The display is adequate for examining output from *neqn*.

DIAGNOSTICS

The exit codes are **0** for normal termination, **2** for all errors.

line too long the representation of a line exceeds 1,024 characters.

SEE ALSO

300(1), col(1), eqn(1), greek(1), nroff(1), tbl(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 line-feeds, backspaces) can make text "disappear"; in particular, tables generated by *tbl*(1) that contain vertical lines often are 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 provide numerical superscript characters, no attempt is made to display them.

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.

EXAMPLE

The following allows the proofreading of *nroff* hyphenation in *textfile*.

mm textfile | hyphen

SEE ALSO

mm(1), troff(1).

BUGS

Hyphen can't cope with hyphenated *italic* (i.e., underlined) words; it often misses them completely or mangles them.

Hyphen occasionally gets confused, but with no ill effects other than spurious extra output.

NAME

id – print user and group IDs and names

SYNOPSIS

id

DESCRIPTION

Id writes a message on the standard output giving the user and group IDs and the corresponding names of the invoking process. If the effective and real IDs do not match, both are printed.

SEE ALSO

logname(1), getuid(2).

NAME

ipcrm – remove a message queue, semaphore set or shared memory id

SYNOPSIS

ipcrm [*options*]

DESCRIPTION

Ipcrm removes one or more specified messages, semaphores, 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).

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

ipcs [options]

DESCRIPTION

Ipcs prints 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 one of the options – **q**, – **m**, or – **s** is specified, only information about the indicated facility is printed. If none of the three options is specified, information about all three is printed.

– **b** Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues; size of segments for shared memory; 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; 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; 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*
The argument is taken as the name of an alternate *namelist* (**/unix** is the default).

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 are to be listed.

T (all) Type of 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, interpreted as follows:

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 disappears 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 follows:

- **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

SEE ALSO

msgop(2), semop(2), shmop(2).

BUGS

The report *ipcs* produces is only a close approximation of the real status, since information can be changed while the program is running.

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.

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*.

Fields are normally separated by a blank, tab, or new-line. In this case, multiple separators count as one, and leading separators are discarded.

These 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.

– **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.

– **t***c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

awk(1), comm(1), sort(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*(1), *comm*(1), *uniq*(1), and *awk*(1) are wildly incongruous.

NAME

kill – terminate a process

SYNOPSIS

kill [– signo] PID ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. This normally kills 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 termination process 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 the first argument, that signal is sent instead of terminate (see *signal*(2)). In particular, the command **kill – 9 ...** is a sure kill.

SEE ALSO

ps(1), sh(1), kill(2), signal(2).

NAME

last – indicate last logins of users and teletypes

SYNOPSIS

last [– N] [name ...] [tty ...]

DESCRIPTION

Last will look back in the *wtmp* file which records all logins and logouts for information about a user, a teletype or any group of users and teletypes. Arguments specify names of users or teletypes of interest. Names of teletypes may be given fully or abbreviated. For example 'last 0' is the same as 'last tty0'. If multiple arguments are given, the information which applies to any of the arguments is printed. For example 'last root console' would list all of "root's" sessions as well as all sessions on the console terminal. *Last* will print the sessions of the specified users and teletypes, most recent first, indicating the times at which the session began, the duration of the session, and the teletype which the session took place on. If the session is still continuing or was cut short by a reboot, *last* so indicates.

The pseudo-user **reboot** logs in at reboots of the system, thus

last reboot

will give an indication of mean time between reboot.

Last with no arguments prints a record of all logins and logouts, in reverse order. The – N option limits the report to N lines.

If *last* is interrupted, it indicates how far the search has progressed in *wtmp*. If interrupted with a quit signal (generated by a control-\) *last* indicates how far the search has progressed so far, and the search continues.

FILES

/usr/adm/wtmp login data base
/usr/adm/shutdownlog which records shutdowns and reasons for same

SEE ALSO

wtmp(5), ac(8), lastcomm(1)

AUTHOR

Howard Katseff

NAME

ld – link editor for common object files

SYNOPSIS

ld [– **a**] [– **e** epsym] [– **f** fill] [– lx] [– **m**] [– **r**] [– **s**] [– **o** outfile] [– **u** symname] [– **L** dir] [– **N**] [– **V**] [– **VS** num] filenames

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 them, producing an object module that can either be executed or used as input for a subsequent *ld* run. The output of *ld* is left in *a.out* . This file is executable if no errors occur during the load. If any input file, *filename*, is not an object file, *ld* assumes it is either a text file containing link editor directives or an archive library.

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 that can be satisfied by library members. Thus, the ordering of library members is unimportant.

The following options are recognized by *ld*.

– **a** Produce an absolute file; give warnings for undefined references. Relocation information is stripped from the output object file unless the – **r** option is given. The – **r** option is needed only when an absolute file should retain its relocation information (not the normal case). If neither – **a** nor – **r** is given, – **a** is assumed.

– **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 two-byte constant.

– l*x* Search a library **lib***x***.a**, where *x* is up to seven characters. A library is searched when its name is encountered, so the placement of a – **l** is significant. By default, libraries are located in **/lib**.

– **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. Unless – **a** is also given, the link editor does not complain about unresolved references.

– **s** Strip line number entries and symbol table information from the output object file.

– **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.

– **L** *dir* Change the algorithm of searching for **lib***x***.a** to look in *dir* before looking in **/lib** and **/usr/lib**. This option is effective only if it precedes the – **l** option on the command line.

– **N** Put the data section immediately following the text in the output file.

-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.

The following information about section alignment and MMU requirements should be considered at system installation.

The default section alignment action for *ld* on M68000 systems is to align the code (**.text**) and data (**.data** and **.bss** combined) separately on 512-byte boundaries. Since MMU requirements vary from system to system, this alignment is not always desirable. The version of *ld* for M68000 systems, therefore, provides a mechanism to allow the specification of different section alignments for each system.

When all input files have been processed (and if no override is provided), *ld* will search the list of library directories (as with the **-l** option) for a file named **default.ld**. If this file is found, it is processed as an *ld* instruction file (or **ifile**). The **default.ld** file should specify the required alignment as outlined below. If it does not exist, the default alignment action will be taken.

The **default.ld** file should appear as follows, with <*alignment*> replaced by the alignment requirement in bytes:

```
SECTIONS {
        .text : {}
        GROUP ALIGN( <alignment>) : {
                        .data : {}
                        .bss  : {}
                        }
        }.
```

For example, a **default.ld** file of the following form would provide the same alignment as the default (512-byte boundary):

```
SECTIONS {
        .text :. {}
        GROUP ALIGN(512) : {
                        .data : {}
                        .bss  : {}
                        }
        }
```

To get alignment on 2K-byte boundaries, the following **default.ld** file would be specified:

```
SECTIONS {
        .text : {}
        GROUP ALIGN(2048) : {
                        .data : {}
                        .bss : {}
                        }
        }
```

For more information about the format of *ld* instruction files or the meaning of the commands, see the "Common Link Editor Reference Manual."

FILES

/lib	
/usr/lib	
a.out	output file

SEE ALSO

as(1), cc(1), a.out(4), ar(4).

WARNINGS

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 should insure 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 data 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 insure that the program always calls exit( ) rather than falling through the end of the entry routine.

NAME

leave – remind you when you have to leave

SYNOPSIS

leave [hhmm]

DESCRIPTION

Leave waits until the specified time, then reminds you that you have to leave. You are reminded 5 minutes and 1 minute before the actual time, at the time, and every minute thereafter. When you log off, *leave* exits just before it would have printed the next message.

The time of day is in the form hhmm where hh is a time in hours (on a 12 or 24 hour clock). All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

If no argument is given, *leave* prompts with "When do you have to leave?". A reply of newline causes *leave* to exit, otherwise the reply is assumed to be a time. This form is suitable for inclusion in a *.login* or *.profile*.

Leave ignores interrupts, quits, and terminates. To get rid of it you should either log off or use "kill – 9" giving its process id.

SEE ALSO

calendar(1)

AUTHOR

Mark Horton

BUGS

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [– **rctvn**] [file] ...

DESCRIPTION

Lex 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 occurrences of, the previous character or character class. The character . is the class of all ASCII characters except new-line. 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. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character $ at the end of an expression requires a trailing new-line. 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 \. Thus [**a**– **zA**– **Z**]+ matches a string of letters.

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");
"/*"    {       loop:
                while (input() != '*');
                switch (input())
```

```
        {
        case '/': break;
        case '*': unput('*');
        default: go to loop;
        }
}
```

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 of the machine generated, **– n** causes the **–** summary not to print. 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 2000)

%n *n* number of states is *n* (500)

%t *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions 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).
"LEX - Lexical Analyzer Generator" by M. E. Lesk and E. Schmidt.
"Lexical Analyzer Generator (LEX)" in the *Support Tools Guide*.

BUGS

The **– r** option is not yet fully operational.

NAME

lfnt – load font

SYNOPSIS

lfnt [fontnum] [fontname] [window]

DESCRIPTION

Lfnt loads the font in file *fontname* into window *window* and assigns it number *fontnum*. If *window* is not supplied, it defaults to the current window. *Fontname* must be a complete pathname of the font file. *Fontnum* must be in the range 0 to 6 (7 is the default font and is automatically loaded).

FILES

/Fonts/CRT contains available fonts.

SEE ALSO

lsfnt(1) sfnt(1) cfnt(1)

NAME

lid, gid, eid - query id database

SYNOPSIS

lid [- **f**file] [- **un**] [- **medoxa**] patterns...

gid [- **f**file] [- **mdoxa**] patterns...

eid [- **f**file] [- **mdoxa**] patterns...

DESCRIPTION

These commands provide a flexible query interface to the *id* database. *Lid* does a lookup on *patters* and prints out lines in this way:

idname ../hdir/hfile.h ../cdir/{cfile1,cfile2}.c

Notice that multiple files with the same directory prefix and suffix are concatenated in the globbing-set-notation of *csh*(1). Also notice that all of the *id* database query commands adjust the list of pathnames to be relative to your current working directory, provided that *mkid*(1) was used to build the database, and your working directory is located within the sub-tree covered by the *id* database.

If multiple names match on pattern, then there will be one line of output per name. The mnemonic significance of the name is *l(ookup)id*.

Gid does a lookup and then searches for the names it matches in the files where they occur. The mnemonic for this name is *g(rep)id*.

Eid does a lookup, and then invokes an editor on all files with the matched name as an initial search string. Of course, this name stands for *e(dit)id*.

Patterns may be simple alpha-numeric strings, or regular expressions in the style of *regcmp*(3). If the string contains no regular expression meta-characters, it is searched for as a *word*. If the string contains meta-characters, or if the - e argument is supplied, it is searched for as regular-expression.

The following options are recognized: The - **f** option, followed immediately by a file name may be used to specify a particular file to be used as the *id* database.

The - **u** option lists all identifiers in the database that are non-unique within the first - *n* characters.

The remaining options are for use in conjunction with numeric patterns. The - **d** option limits numeric matches to the decimal radix. The - **o** and - **x** options limit matches to octal and hexadecimal respectively. More than one radix option may be provided, and - **a** is a shorthand for specifying all three.

Searches for numeric *id*s are done numerically rather than lexically, so that all representations for a given number are potentially available in a single search.

SEE ALSO

mkid(1), fid(1).

BUGS

Eid should use the **EDITOR** environment variable rather than always using *vi*(1).

NAME

line – read one line

SYNOPSIS

line

DESCRIPTION

Line copies one line (up to a new-line) 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 new-line. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1), read(2).

NAME

lint – a C program checker

SYNOPSIS

lint [– **abhlnpuvx**] file ...

DESCRIPTION

Lint attempts to detect features of the C program *files* which are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the features 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 which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

It is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. By default, *lint* uses function definitions from the standard lint library **llib-lc.ln**; function definitions from the portable lint library **llib-port.ln** are used when *lint* is invoked with the – **p** option.

Any number of *lint* options may be used, in any order. 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* often result in a large number of 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* behavior:

– **l***x* Include additional lint library **llib-l***x***.ln**. 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**. This option can be used to keep 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.

The – **D**, – **U**, and – **I** options of *cc*(1) are also recognized as separate arguments.

Certain conventional comments in the C source change the behavior of *lint*:

/*NOTREACHED*/
at appropriate points stops comments about unreachable code.

/*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 in the file.

Lint produces its first output on a per source file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, 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 filename is printed, followed by a question mark.

FILES

/usr/lib/lint[12]	programs
/usr/lib/llib-lc.ln	declarations for standard functions (binary format; source is in **/usr/lib/llib-lc**)
/usr/lib/llib-port.ln	declarations for portable functions (binary format; source is in **/usr/lib/llib-port**)
/usr/lib/llib-lm.ln	declarations for standard math functions (binary format; source is in **/usr/lib/llib-lm**)
/usr/tmp/*lint*	temporaries

SEE ALSO

cc(1).
"A C Program Checker - lint" in the *Programming Guide*.

BUGS

Exit(2) and other functions which do not return are not understood; this causes various inaccuracies.

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. It is invoked by the system when a previous user has terminated the initial shell by typing a *cntrl-d* to indicate an end-of-file. (See "How to Get Started" at the beginning of this volume for instructions on how to dial up initially.)

If *login* is invoked as a command it must replace the initial command interpreter. This is accomplished by typing:

exec login

from the initial shell.

Login asks for your user name (if not supplied as an argument), and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it does not appear on the written record of the session.

At some installations, an option may be invoked that requires you to enter a second "dialup" password. This occurs only for dial-up connections, and is prompted by the message "dialup password:". Both passwords are required for a successful login.

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)) are 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.

The basic *environment* (see *environ*(5)) 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, then 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 which aren't 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*
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/profile	system profile
.profile	user's login profile

SEE ALSO

mail(1), newgrp(1), sh(1), su(1), passwd(4), profile(4), environ(5).

DIAGNOSTICS

Login incorrect The user name or password cannot be matched.

No shell, cannot open password file, or **no directory**.
Consult a system programming counselor.

No utmp entry. You must exec "login" from the lowest level "sh".
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).

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 filenames; 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 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

*symref, *symdef temporary files

SEE ALSO

ar(1), ld(1), tsort(1), ar(4).

BUGS

Object files whose names do not end with **.o**, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

NAME

lp, cancel – send/cancel requests to an LP line printer

SYNOPSIS

lp [– **c**] [– **d***dest*] [– **m**] [– **n***number*] [– **o***option*] [– **s**] [– **t***title*] [– **w**] files
cancel [ids] [printers]

DESCRIPTION

Lp arranges for the named files and associated information (collectively called a *request*) to be printed by a line printer. If no filenames are specified, the standard input is assumed. The filename – stands for the standard input and may be supplied on the command line in conjunction with named *files*. The order in which *files* are specified is the same order in which they are printed.

Lp associates a unique *id* with each request and prints it on the standard output. This *id* can be used later to cancel (see *cancel*) or find the status (see *lpstat*(1)) of the request.

The following options to *lp* may appear in any order and may be intermixed with filenames:

– **c** Make copies of the *files* to be printed immediately when *lp* is invoked. Normally, *files* are not copied, but are linked whenever possible. If the – **c** option is not given, then the user should be careful not to remove any of the *files* before the request has been printed in its entirety. It should also be noted that 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* Choose *dest* as the printer or class of printers that is to do the printing. If *dest* is a printer, then the request is printed only on that specific printer. If *dest* is a class of printers, then the request is printed on the first available printer that is a member of the class. Under certain conditions (e.g., printer unavailability, file space limitation), 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 between systems (see *lpstat*(1)).

– **m** Send mail (see *mail(1)*) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

– **n***number* Print *number* copies (default of 1) of the output.

– **o***option* Specify printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the – **o** keyletter more than once. For more information about what is valid for *options*, see *Models* in *lpadmin*(1M).

– **s** Suppress messages from *lp*(1) such as "request id is ...".

– **t***title* Print *title* on the banner page of the output.

– **w** Write a message on the user's terminal after the *files* have been printed. If the user is not logged in, then mail is sent instead.

Cancel cancels line printer requests made by the *lp*(1) command. The 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.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lpstat(1), mail(1).
accept(1M), lpadmin(1M), lpsched(1M) in the *Administrator's Manual*.

NAME

lpd – line printer daemon

SYNOPSIS

/usr/lib/lpd

DESCRIPTION

Lpd is the daemon for a line printer and uses the directory **/usr/spool/lpd**. The file **lock** in either directory is used to prevent two daemons from becoming active simultaneously. After the program has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with "df". Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

L specifies that the remainder of the line is to be sent as a literal.

I is the same as **L**, but signals the $ IDENT card which is to be mailed back by the mail option.

B specifies that the rest of the line is a filename. That file is to be sent as binary cards.

F is the same as **B** except a form-feed is prepended to the file.

U specifies that the rest of the line is a filename. After the job has been transmitted, the file is unlinked.

M is followed by a user ID; after the job is sent, a message is mailed to the user via the *mail*(1) command to verify the sending of the job.

N is followed by a user filename, to be sent back under the mail option.

Any error encountered causes the daemon to drop the call, wait up to 10 seconds, and start over. This means that an improperly constructed "df" file may cause the same job to be submitted every 10 seconds.

Lpd is automatically initiated by the line printer command, *lpr*.

To restart *lpd* (in the case of hardware or software malfunction), it is necessary to first kill the old daemon (if it is still alive), and remove the lock file (if present), before initiating the new daemon. This can be done automatically by **/etc/rc** when the system is brought up, in the event there were jobs left in the spooling directory when the system last went down.

FILES

/usr/spool/lpd/*	spool area for line printer daemon.
/etc/passwd	to get the user's name.
/dev/lp	line printer device.

SEE ALSO

lpr(1).

BUGS

If a *umask*(1) of 077 is used, the print jobs may be spooled but cannot be printed.

NAME

lpr – line printer spooler

SYNOPSIS

lpr [option ...] [name ...]

DESCRIPTION

Lpr causes the named files to be queued for printing on a line printer. If no names appear, the standard input is assumed; thus *lpr* may be used as a filter.

The following *options* may be given (each as a separate argument and in any order) before any filename arguments:

- **–c** Make a copy of the file to be sent before returning to the user.
- **–r** Remove the file after sending it.
- **–m** Report by *mail*(1) when printing is complete. *mail*(1).
- **–n** Do not report the completion of printing by *mail*(1). This is the default option.
- **–f***file* Use *file* as a dummy filename to report back by *mail*(1). (This is useful for distinguishing multiple runs, especially when *lpr* is being used as a filter).

FILES

/etc/passwd	user's identification and accounting data.
/usr/lib/lpd	line printer daemon.
/usr/spool/lpd/*	spool area.

SEE ALSO

lpd(1C), lp(1).

NAME

lpstat – print LP status information

SYNOPSIS

lpstat [options]

DESCRIPTION

Lpstat prints information about the current status of the LP line printer system.

If no *options* are given, then *lpstat* prints the status of all requests made to *lp*(1) by the user. Any arguments that are not *options* are assumed to be request *ids* (as returned by *lp*). *Lpstat* prints the status of such requests. *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""

The omission of a *list* following such keyletters causes all information relevant to the keyletter to be printed. For example,

lpstat – o

prints the status of all output requests.

– a[*list*] Print acceptance status (with respect to *lp*) of destinations for requests. *List* is a list of intermixed printer names and class names.

– c[*list*] Print class names and their members. *List* is a list of class names.

– d Print the system default destination for *lp*.

– o[*list*] Print the status of output requests. *List* is a list of intermixed printer names, class names, and request *ids*.

– p[*list*] Print the status of printers. *List* is a list of printer names.

– r Print the status of the LP request scheduler

– s Print a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, and a list of printers and their associated devices.

– t Print all status information.

– u[*list*] Print status of output requests for users. *List* is a list of login names.

– v[*list*] Print the names of printers and the pathnames of the devices associated with them. *List* is a list of printer names.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1).

NAME

ls – list contents of directories

SYNOPSIS

ls [– **logtasdrucifp**] names

DESCRIPTION

Ls lists the contents of each named directory; for each file named, *ls* repeats the filename and any other information requested. By default, the output is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments are processed before directories and their contents. There are several options:

– **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 contains the major and minor device numbers, rather than a size.

– **o** The same as – **l**, except that the group is not printed.

– **g** The same as – **l**, except that the owner is not printed.

– **t** Sort by time of last modification (latest first) instead of by name.

– **a** List all entries; in the absence of this option, entries whose names begin with a period (**.**) are *not* listed.

– **s** Give size in blocks (including indirect blocks) for each entry.

– **d** If argument is a directory, list only its name; often used with – **l** to get the status of a directory.

– **r** Reverse the order of sort to get reverse alphabetic or oldest first, as appropriate.

– **u** Use time of last access instead of last modification for sorting (with the – **t** option) and/or printing (with the – **l** option).

– **c** Use time of last modification of the inode (mode, etc.) instead of last modification of the file for sorting (– **t**) and/or printing (– **l**).

– **i** For each file, print the i-number in the first column of the report.

– **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.

– **p** Put a slash after each filename if that file is a directory. Especially useful for CRT terminals when combined with the *pr*(1) command as follows: **ls – p | pr – 5 – t – w80**.

The mode printed under the – **l** option consists of 11 characters that are interpreted as follows:

The first character is:

d if the entry is a directory;
b if the entry is a block special file;
c if the entry is a character special file;
p if the entry is a fifo (a.k.a. "named pipe") special file;
– if 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.

The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
– if the indicated permission is *not* granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise, the user-execute permission character is given as **s** if the file has set-user-ID mode. The last character of the mode (normally **x** or –) is **t** if the 1000 (octal) bit of the mode is on; see *chmod*(1) for the meaning of this mode. The indications of set-ID and 1000 bit of the mode are capitalized (**S** and **T** respectively) if the corresponding execute permission is *not* set.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

FILES

/etc/passwd	to get user IDs for **ls – l** and **ls – o**.
/etc/group	to get group IDs for **ls – l** and **ls – g**.

SEE ALSO

chmod(1), find(1).

NAME

ls – list contents of directory

SYNOPSIS

ls [– **acdfgilqrstu1ACLFR**] name ...

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. By default, the output is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments are processed before directories and their contents.

There are a large number of options:

– **l** List in long format, giving mode, number of links, owner, 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. If the file is a symbolic link the pathname of the linked-to file is printed preceded by "– >".

– **g** Include the group ownership of the file in a long output.

– **t** Sort by time modified (latest first) instead of by name.

– **a** List all entries; in the absence of this option, entries whose names begin with a period (.) are *not* listed.

– **s** Give size in kilobytes of each file.

– **d** If argument is a directory, list only its name; often used with – **l** to get the status of a directory.

– **L** If argument is a symbolic link, list the file or directory the link references rather than the link itself.

– **r** Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

– **u** Use time of last access instead of last modification for sorting (with the – **t** option) and/or printing (with the – l option).

– **c** Use time of file creation for sorting or printing.

– **i** For each file, print the i-number in the first column of the report.

– **f** Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off – **l,** – **t,** – **s,** and – **r,** and turns on – **a;** the order is the order in which entries appear in the directory.

– **F** cause directories to be marked with a trailing '/', sockets with a trailing '=', symbolic links with a trailing '@', and executable files with a trailing '*'.

– **R** recursively list subdirectories encountered.

– **1** force one entry per line output format; this is the default when output is not to a terminal.

– **C** force multi-column output; this is the default when output is to a terminal.

– **q** force printing of non-graphic characters in file names as the character '?'; this is the default when output is to a terminal.

The mode printed under the – l option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;

b if the entry is a block-type special file;
c if the entry is a character-type special file;
l if the entry is a symbolic link;
s if the entry is a socket, or
– if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
– if the indicated permission is not granted.

The group-execute permission character is given as **s** if the file has the set-group-id bit set; likewise the user-execute permission character is given as **s** if the file has the set-user-id bit set.

The last character of the mode (normally 'x' or '– ') is **t** if the 1000 bit of the mode is on. See *chmod*(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user id's for 'ls – l'.
/etc/group to get group id's for 'ls – g'.

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "ls – s" is much different than "ls – s |lpr". On the other hand, not doing this setting would make old shell scripts which used *ls* almost certain losers.

NAME

lsfnt – list loaded fonts

SYNOPSIS

lsfnt

DESCRIPTION

Lsfnt Displays a list of all loaded fonts for the window. The font number is listed (* next to the font currently selected) along with the pathname of the loaded font. The pathname is displayed in its own font type.

SEE ALSO

lfnt(1) sfnt(1) cfnt(1)

NAME

m4 – macro processor

SYNOPSIS

m4 [options] [files]

DESCRIPTION

M4 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 filename 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 filenames and before any **– D** or **– U** flags:

– D*name*[=*val*]
Defines *name* to *val* or to null in the absence of *val*.

– 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 assumed 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.

M4 ignores leading unquoted blanks, tabs, and new-lines 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 that appear 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-in ones.

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 the 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 is subsequently performed.

changequote changes 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 changes left and right comment markers from the default # and new-line. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes new-line. 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 from all diversions if no argument is present. 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 new-line.

ifelse has three or more arguments. If the first argument is the same string as the second, then 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 system 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	pushes back argument 1 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 built-in ones); otherwise, it 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).
"The M4 Macro Processor" by B. W. Kernighan and D. M. Ritchie.
"The M4 Macro Processor" in the *Support Tools Guide*.

NAME

pdp11, u3b, vax, m68k – provide truth value about your processor type

SYNOPSIS

pdp11

u3b

vax

m68k

DESCRIPTION

The following commands return a true value (exit code of 0) if you are on the processor that the command name indicates.

pdp11 True if you are on a PDP-11/45 or PDP-11/70.

u3b True if you are on a 3B20S.

vax True if you are on a VAX-11/750 or VAX-11/780.

m68k True if you are on a Motorola M68000 processor.

The commands that do not apply return a false (non-zero) value. These commands are often used within *make*(1) makefiles and shell procedures to increase portability.

SEE ALSO

sh(1), test(1), true(1).

NAME

mail, rmail – send mail to users or read mail

SYNOPSIS

mail [**– epqr**] [**– f** file]

mail [**– t**] persons

rmail [**– t**] persons

DESCRIPTION

Mail without arguments prints a user's mail, message-by-message, in last-in, first-out order. For each message, the user is prompted with a **?**, and a line is read from the standard input to determine the disposition of the message:

<new-line>	Go on to next message.
+	Same as <new-line>.
d	Delete message and go on to next message.
p	Print message again.
–	Go back to previous message.
s [*files*]	Save message in the named *files* (**mbox** is default).
w [*files*]	Save message, without its header, in the named *files* (**mbox** is default).
m [*persons*]	Mail the message to the named *persons* (yourself is default).
q	Put undeleted mail back in the *mailfile* and stop.
EOT (control-d)	Same as **q**.
x	Put all mail back in the *mailfile* unchanged and stop.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

The optional arguments alter the printing of the 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.

– **p** causes all mail to be printed without prompting for disposition.

– **q** causes *mail* to terminate after interrupts. Normally an interrupt only causes 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*.

When *persons* are named, *mail* takes the standard input up to an end-of-file (or up to a line consisting of just a **.**) and adds it to each *person*'s *mailfile*. The message is preceded by the sender's name and a postmark. Lines that look like postmarks in the message, (i.e., "From ...") are preceded with a >. The **– t** option causes the message to be preceded by all *persons* the *mail* is sent to. A *person* is usually a user name recognized by *login*(1). If a *person* being sent mail is not recognized, or if *mail* is interrupted during input, the file **dead.letter** is saved to allow editing and resending.

To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1C)). Everything after the first exclamation mark in *persons* is interpreted by the remote system. In particular, if *persons* contains additional exclamation marks, it can denote a sequence of machines through which the message is to be sent on the way to its ultimate destination. For example, specifying **a!b!cde** as a recipient's name causes the message to be sent to user **b!cde** on system **a**. System **a** interprets that destination as a request to send the message to user **cde** on system **b**. This might be useful, for instance, if the sending system can access system **a** but not system **b**, and system **a** has access to system **b**.

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 is preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

Forward to *person*

which causes all mail sent to the owner of the *mailfile* to be forwarded to *person*. This is especially useful to forward all of a person's mail to one machine in a multiple machine environment.

Rmail only permits the sending of mail; *uucp*(1C) uses *rmail* as a security precaution.

When a user logs in, the presence of mail, if any, is indicated. Also, notification is made if new mail arrives while using *mail*.

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), uucp(1C), write(1).

BUGS

Race 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

mail – send and receive mail

SYNOPSIS

mail [**–v**] [**–i**] [**–n**] [**–s** subject] [user ...]
mail [**–v**] [**–i**] [**–n**] **–f** [name]
mail [**–v**] [**–i**] [**–n**] **–u** user

INTRODUCTION

Mail is a intelligent mail processing system, which has a command syntax reminiscent of *ed* with lines replaced by messages.

The **–v** flag puts mail into verbose mode; the details of delivery are displayed on the users terminal. The **–i** flag causes tty interrupt signals to be ignored. This is particularly useful when using *mail* on noisy phone lines. The **–n** flag inhibits the reading of /usr/lib/Mail.rc.

Sending mail. To send a message to one or more other people, *mail* can be invoked with arguments which are the names of people to send to. You are then expected to type in your message, followed by an EOT (control– D) at the beginning of a line. A subject may be specified on the command line by using the **–s** flag. (Only the first argument after the **–s** flag is used as a subject; be careful to quote subjects containing spaces.) The section below, labeled *Replying to or originating mail,* describes some features of *mail* available to help you compose your letter.

Reading mail. In normal usage *mail* is given no arguments and checks your mail out of the post office, then prints out a one line header of each message there. The current message is initially the first message (numbered 1) and can be printed using the **print** command (which can be abbreviated **p**). You can move among the messages much as you move between lines in *ed*, with the commands '+' and '–' moving backwards and forwards, and simple numbers.

Disposing of mail. After examining a message you can **delete** (**d**) the message or **reply** (**r**) to it. Deletion causes the *mail* program to forget about the message. This is not irreversible; the message can be **undeleted** (**u**) by giving its number, or the *mail* session can be aborted by giving the **exit** (**x**) command. Deleted messages will, however, usually disappear never to be seen again.

Specifying messages. Commands such as **print** and **delete** can be given a list of message numbers as arguments to apply to a number of messages at once. Thus "delete 1 2" deletes messages 1 and 2, while "delete 1– 5" deletes messages 1 through 5. The special name "*" addresses all messages, and "$" addresses the last message; thus the command **top** which prints the first few lines of a message could be used in "top *" to print the first few lines of all messages.

Replying to or originating mail. You can use the **reply** command to set up a response to a message, sending it back to the person who it was from. Text you then type in, up to an end-of-file, defines the contents of the message. While you are composing a message, *mail* treats lines beginning with the character "~" specially. For instance, typing "~m" (alone on a line) will place a copy of the current message into the response right shifting it by a tabstop. Other escapes will set up subject fields, add and delete recipients to the message and allow you to escape to an editor to revise the message or to a shell to run some commands. (These options are given in the summary below.)

Ending a mail processing session. You can end a *mail* session with the **quit** (**q**) command. Messages which have been examined go to your *mbox* file unless they have been deleted in which case they are discarded. Unexamined messages go back to the post office. The **–f** option causes *mail* to read in the contents of your *mbox* (or the specified file) for processing; when you **quit**, *mail* writes undeleted messages back to this file. The **–u** flag is a short way of doing "mail **–f** /usr/spool/mail/user".

Personal and systemwide distribution lists. It is also possible to create a personal distribution lists so that, for instance, you can send mail to "cohorts" and have it go to a group of people. Such lists can be defined by placing a line like

alias cohorts bill ozalp jkf mark kridle@ucbcory

in the file .mailrc in your home directory. The current list of such aliases can be displayed with the **alias (a)** command in *mail*. System wide distribution lists can be created by editing /usr/lib/aliases, see *aliases*(5) and *sendmail*(8); these are kept in a different syntax. In mail you send, personal aliases will be expanded in mail sent to others so that they will be able to **reply** to the recipients. System wide *aliases* are not expanded when the mail is sent, but any reply returned to the machine will have the system wide alias expanded as all mail goes through *sendmail*.

Network mail (ARPA, UUCP, Berknet) See *mailaddr(7)* for a description of network addresses.

Mail has a number of options which can be set in the *.mailrc* file to alter its behavior; thus "set askcc" enables the "askcc" feature. (These options are summarized below.)

SUMMARY

(Adapted from the 'Mail Reference Manual')

Each command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety – the first command which matches the typed prefix is used. For commands which take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, *mail* types "No applicable messages" and aborts the command.

–	Goes to the previous message and prints it out. If given a numeric argument *n*, goes to the *n*-th previous message and prints it.
?	Prints a brief summary of commands.
!	Executes the UNIX shell command which follows.
Print	(**P**) Like **print** but also prints out ignored header fields. See also **print** and **ignore.**
Reply	(R) Reply to originator. Does not reply to other recipients of the original message.
Type	(T) Identical to the **Print** command.
alias	(**a**) With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, creates an new or changes an on old alias.
alternates	(**alt**) The **alternates** command is useful if you have accounts on several machines. It can be used to inform *mail* that the listed addresses are really you. When you **reply** to messages, *mail* will not send a copy of the message to any of the addresses listed on the *alternates* list. If the **alternates** command is given with no argument, the current set of alternate names is displayed.
chdir	(**c**) Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.
copy	(**co**) The **copy** command does the same thing that **save** does, except that it does not mark the messages it is used on for deletion when you quit.
delete	(**d**) Takes a list of messages as argument and marks them all as deleted. Deleted messages will not be saved in *mbox*, nor will they be available for most other commands.

dp (also **dt**) Deletes the current message and prints the next message. If there is no next message, *mail* says "at EOF."

edit (**e**) Takes a list of messages and points the text editor at each one in turn. On return from the editor, the message is read back in.

exit (**ex** or **x**) Effects an immediate return to the Shell without modifying the user's system mailbox, his *mbox* file, or his edit file in **– f**.

file (**fi**) The same as **folder**.

folders List the names of the folders in your folder directory.

folder (**fo**) The **folder** command switches to a new mail file or folder. With no arguments, it tells you which file you are currently reading. If you give it an argument, it will write out changes (such as deletions) you have made in the current file and read in the new file. Some special conventions are recognized for the name. # means the previous file, % means your system mailbox, %user means user's system mailbox, & means your ~/mbox file, and +folder means a file in your folder directory.

from (**f**) Takes a list of messages and prints their message headers.

headers (**h**) Lists the current range of headers, which is an 18 message group. If a "+" argument is given, then the next 18 message group is printed, and if a "–" argument is given, the previous 18 message group is printed.

help A synonym for ?

hold (**ho**, also **preserve**) Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in *mbox*. Does not override the **delete** command.

ignore Add the list of header fields named to the *ignored list*. Header fields in the ignore list are not printed on your terminal when you print a message. This command is very handy for suppression of certain machine-generated header fields. The **Type** and **Print** commands can be used to print a message in its entirety, including ignored fields. If **ignore** is executed with no arguments, it lists the current set of ignored fields.

mail (**m**) Takes as argument login names and distribution group names and sends mail to those people.

mbox Indicate that a list of messages be sent to *mbox* in your home directory when you quit. This is the default action for messages if you do *not* have the *hold* option set.

next (**n** like + or CR) Goes to the next message in sequence and types it. With an argument list, types the next matching message.

preserve (**pre**) A synonym for **hold**.

print (**p**) Takes a message list and types out each message on the user's terminal.

quit (**q**) Terminates the session, saving all undeleted, unsaved messages in the user's *mbox* file in his login directory, preserving all messages marked with **hold** or **preserve** or never referenced in his system mailbox, and removing all other messages from his system mailbox. If new mail has arrived during the session, the message "You have new mail" is given. If given while editing a mailbox file with the **– f** flag, then the edit file is rewritten. A return to the Shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.

reply (**r**) Takes a message list and sends mail to the sender and all recipients of the specified message. The default message must not be deleted.

respond A synonym for **reply**.

save (**s**) Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal.

set (**se**) With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form "option=value" or "option."

shell (**sh**) Invokes an interactive version of the shell.

size Takes a message list and prints out the size in characters of each message.

source (**so**) The **source** command reads *mail* commands from a file.

top Takes a message list and prints the top few lines of each. The number of lines printed is controlled by the variable **toplines** and defaults to five.

type (**t**) A synonym for **print**.

unalias Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any significance.

undelete (**u**) Takes a message list and marks each one as *not* being deleted.

unset Takes a list of option names and discards their remembered values; the inverse of **set**.

visual (**v**) Takes a message list and invokes the display editor on each message.

write (**w**) A synonym for **save**.

xit (**x**) A synonym for **exit**.

z *Mail* presents message headers in windowfuls as described under the **headers** command. You can move *mail*'s attention forward to the next window with the **z** command. Also, you can move to the previous window by using **z–** .

Here is a summary of the tilde escapes, which are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option **escape**.

~!command Execute the indicated shell command, then return to the message.

~c name ... Add the given names to the list of carbon copy recipients.

~d Read the file "dead.letter" from your home directory into the message.

~e Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.

~f messages Read the named messages into the message being sent. If no messages are specified, read in the current message.

~h Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.

~m messages
Read the named messages into the message being sent, shifted right one tab. If no messages are specified, read the current message.

~**p**	Print out the message collected so far, prefaced by the message header fields.
~**q**	Abort the message being sent, copying the message to "dead.letter" in your home directory if **save** is set.
~**r** filename	Read the named file into the message.
~**s** string	Cause the named string to become the current subject field.
~**t** name ...	Add the given names to the direct recipient list.
~**v**	Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor will be a screen editor. After you quit the editor, you may resume appending text to the end of your message.
~**w** filename	Write the message onto the named file.
~|command	Pipe the message through the command as a filter. If the command gives no output or terminates abnormally, retain the original text of the message. The command *fmt*(1) is often used as *command* to rejustify the message.
~~string	Insert the string of text in the message prefaced by a single ~. If you have changed the escape character, then you should double that character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not, or string, in which case the actual value is of interest. The binary options include the following:

append	Causes messages saved in *mbox* to be appended to the end rather than prepended. (This is set in /usr/lib/Mail.rc on version 7 systems.)
ask	Causes *mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.
askcc	Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.
autoprint	Causes the **delete** command to behave like **dp** – thus, after deleting a message, the next one will be typed automatically.
debug	Setting the binary option *debug* is the same as specifying – **d** on the command line and causes *mail* to output all sorts of information useful for debugging *mail*.
dot	The binary option *dot* causes *mail* to interpret a period alone on a line as the terminator of a message you are sending.
hold	This option is used to hold messages in the system mailbox by default.
ignore	Causes interrupt signals from your terminal to be ignored and echoed as @'s.
ignoreeof	An option related to *dot* is *ignoreeof* which makes *mail* refuse to accept a control-d as the end of a message. *Ignoreeof* also applies to *mail* command mode.
metoo	Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.
nosave	Normally, when you abort a message with two RUBOUT, *mail* copies the partial letter to the file "dead.letter" in your home directory. Setting the binary option *nosave* prevents this.

quiet Suppresses the printing of the version when first invoked.

verbose Setting the option *verbose* is the same as using the **– v** flag on the command line. When mail runs in verbose mode, the actual delivery of messages is displayed on he users terminal.

The following options have string values:

EDITOR Pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then a default editor is used.

SHELL Pathname of the shell to use in the **!** command and the ~! escape. A default shell is used if this option is not defined.

VISUAL Pathname of the text editor to use in the **visual** command and ~v escape.

crt The valued option *crt* is used as a threshold to determine how long a message must be before *more* is used to read it.

escape If defined, the first character of this option gives the character to use in the place of ~ to denote escapes.

folder The name of the directory to use for storing folders of messages. If this name begins with a '/', *mail* considers it to be an absolute pathname; otherwise, the folder directory is found relative to your home directory.

record If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not so saved.

toplines If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

FILES

/usr/spool/mail/*	post office
~/mbox	your old mail
~/.mailrc	file giving initial mail commands
/tmp/R#	temporary for editor escape
/usr/lib/Mail.help*	help files
/usr/lib/Mail.rc	system initialization file
Message*	temporary for editing messages

SEE ALSO

binmail(1), fmt(1), newaliases(1), aliases(5),
mailaddr(7), sendmail(8)
'The Mail Reference Manual'

BUGS

There are many flags that are not documented here. Most are not useful to the general user.
Usually, *mail* is just a link to *Mail*, which can be confusing.

AUTHOR

Kurt Shoens

NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [– **f** makefile] [– **p**] [– **i**] [– **k**] [– **s**] [– **r**] [– **n**] [– **b**] [– **e**] [– **m**] [– **t**] [– **d**] [– **q**] [names]

DESCRIPTION

The *make* program provides a method for maintaining up-to-date versions of programs that result from many operations on a number of files. *Make* can keep track of the sequence of commands that create certain files and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of a program, *make* creates the proper files correctly. It also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

The basic operation of *make* is to find the name of a needed target file in the description, ensure that all of the files on which it depends exist and are up-to-date, and then create the target file if it has not been modified since its generators were. The descriptor file defines the graph of dependencies.

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. A filename of – denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

– **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, 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.

– **m** Print a memory map showing text, data, and stack. This option is a no-operation on systems without the *getu* system call.

– **t** Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

– **d** Debug mode. Print out detailed information on files and times examined.

– **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 are not removed when quit or interrupt is hit.

.SILENT Same effect as the – **s** option.

.IGNORE Same effect as the – **i** option.

The *make* program operates using three sources of information: a user-supplied description file; filenames and "last-modified" times from the file system; built-in rules to bridge some of the gaps.

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**, **s.makefile**, and **s.Makefile** are tried in order. If *makefile* is **–** , the standard input is taken. More than one **–** *makefile* argument pair may appear.

Make updates a target only if it depends on files that 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 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><new-line> sequence. Everything printed by make (except the initial tab) is passed directly to the shell as is. Thus,

```
echo a\
b
```

produces

```
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 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 contains the string **$(MAKE)**, 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. The difference between the old version of *make* and this version is that this version requires all dependency lines to have a (possibly null or implicit) command associated with them. The previous version of *make* assumed that if no command was specified explicitly, the command was null.

Interrupt and quit cause the target to be deleted unless the target is a dependency 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.

The MAKEFLAGS environment variable is processed by *make* as containing any legal input option (except **– f**, **– p**, and **– d**) 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 the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped newline. 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 which 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.

f3 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 which is out of date with respect to the target (i.e., the "manufactured" dependent filename). 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 $?. The reasons for this are debatable.

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~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
```

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
```

The only peculiarity in this output is the (**null**) string which *printf*(3S) prints when handed a null string.

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 .y .l .s
```

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. 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 **$(LIB)(file.o)** both refer to an archive library which contains **file.o**. (This assumes the LIB macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form **.*XX*.a** , where *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires *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 rv $@ $*.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 rv 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, one 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, but becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile and s.[Mm]akefile

SEE ALSO

sh(1).
"Make - A Program for Maintaining Computer Programs" by S. I. Feldman.
"An Augmented Version of Make" by E. G. Bradford.
Support Tools Guide.

BUGS

Some commands return non-zero status inappropriately; use **-i** to overcome the difficulty. 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~)** doesn't work.

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 reads 10 bytes from its standard input, and writes 13 bytes on its standard output. 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 (e.g., *ed*(1) and *crypt*(1)). Usually, the input and output are pipes.

SEE ALSO

crypt(1), ed(1), passwd(4).

NAME

man, manprog – print entries in this manual

SYNOPSIS

man [options] [section] titles

/usr/lib/manprog file

DESCRIPTION

Man locates and prints the 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 lower case. The *section* number may not have a letter suffix. If no *section* is specified, the whole manual is searched for *title* and all occurrences of it are printed. *Options* and their meanings are:

– **t** Typeset the entry in the default format size (8.5″×11″).

– **s** Typeset the entry in the small format size (6″×9″).

– **T4014** Display the typeset output on a Tektronix 4014 terminal using *tc*(1).

– **Ttek** Same as – **T4014**.

– **Tst** Print the typeset output on the MHCC STARE facility; this option is not usable on most systems.

– **Tvp** Print the typeset output on a Versatec printer; this option is not available at all sites.

– **T***term* Format the entry using *nroff*(1) and print it on the standard output (usually, the terminal); *term* is the terminal type (see *term*(5) and the explanation below); 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/man**, or to the current directory for – **d** option.

– **d** Search the current directory rather than **/usr/man**; requires the full filename (e.g., **cu.1c**, rather than just **cu**).

– **12** Indicates that the manual entry is to be produced in 12-pitch. May be used when $TERM (see below) is set to one of **300**, **300s**, **450**, and **1620**. The pitch switch on the DASI 300 and 300s terminals must be manually set to **12** if this option is used.

– **c** Causes *man* to 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**.

– **y** Causes *man* to use the non-compacted version of the macros.

The above *options* other than – **d**, – **c**, and – **y** are mutually exclusive, except that the – **s** option may be used in conjunction with the first 4 – **T** options above. Any other *options* are passed to *troff*(1), *nroff*(1), or the *man*(5) macro package.

When using *nroff*(1), *man* examines the environment variable $TERM (see *environ*(5)) and attempts to select options to *nroff*(1), as well as filters, that adapt the output to the terminal being used. The – **T***term* option overrides the value of $TERM; in particular, one should use

– **Tlp** 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).

If the first line of the input for an entry consists solely of the string:

'\" *x*

where *x* is any combination of the three characters **c**, **e**, and **t**, and where there is exactly one blank between the double quote (") and *x*, then *man* preprocesses its input through the appropriate combination of *cw*(1), *eqn*(1) (*neqn* for *nroff*(1)) and *tbl*(1), respectively; when invoked, *eqn*(1) or *neqn* automatically read the file **/usr/pub/eqnchar** (see *eqnchar*(5)).

The *man* command executes *manprog*, which takes a filename as its argument. *Manprog* calculates and returns a string of three register definitions used by the formatters identifying the date the file was last modified. The returned string has the form:

– **rd***day* – **rm***month* – **ry***year*

and is passed to *nroff*(1), which sets this string as variables for the *man* macro package. Months are given from 0 to 11, therefore month is always 1 less than the actual month. The *man* macros calculate the correct month. If the *man* macro package is invoked as an option to *nroff*/*troff* (i.e., *nroff – man file*), then the current day/month/year is used as the printed date.

FILES

/usr/man/u_man/man[1-6]/*	the User's Manual
/usr/man/a_man/man[178]/*	the Administator's Manual
/usr/man/local/man[1-8]/*	local additions
/usr/lib/manprog	calculates modification dates of entries

SEE ALSO

cw(1), eqn(1), nroff(1), tbl(1), tc(1), troff(1), environ(5), man(5), term(5).

BUGS

All entries are supposed to be reproducible either on a typesetter or on a terminal. However, on a terminal some information is necessarily lost.

If pages bearing the same name appear in both manuals, the *Administrator's Manual* entry is printed first, unless a *section* argument is supplied.

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

mince – video text editor

SYNOPSIS

mince [file]

DESCRIPTION

Mince is a screen-oriented text editor.

If the *file* argument is given, *mince* reads that file into a buffer to be edited. If the file does not exist, it is created as an empty file in the buffer.

Complete documentation of the *mince* command is provided in the **Mince Manual,** included as part of the standard LMI documentation set.

NAME

mkdir – make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777 (possibly altered by *umask*(1)). Standard entries, ., for the directory itself, and .., for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

sh(1), rm(1), umask(1).

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

NAME

mkid – make an id database

SYNOPSIS

mkid [– ofile] [– cc] [– accc] [– **u**] [– **v**] [–] files...

DESCRIPTION

Mkid builds a database that maps identifiers to the files in which they occur. The files that it uses may be specified on the command line, or they may be presented one-per-line on the *standard-input* if the - argument is given in lieu of filenames. If no files are specified on the command line, and no *filesfile* is given, *mkid* will look for an existing database and extract a list of pathnames from there. Moreover, *mkid* will only scan those files that have modification times later than that of the database itself. This provides a less expensive means of keeping an *id* database up-to-date when no pathnames need to be added or deleted from the list of constituent files.

The – **o** option specifies a filename in which to place the finished database. The default name is *id*.

Mkid is designed primarily for use on *C* source code, although it may be used on sources for other languages, and other types of text with varying degrees of success. The features that make *mkid* it suited to *C* are that it ignores *C* style comments, quoted strings, and character constants. It uses the lexical rules of *C* to pick out identifiers, numbers, and pre-processor commands.

Mkid is also suitable for running on assembly language sources. Since assemblers vary widely in their syntax rules, several options are provided to control *mkid*s behavior on these.

Use the – **u** option if your assembler prepends an *underscore* character to external names. *Mkid* will strip the *underscore* from the name before inserting it into the database. This ensures that queries for identifiers that occur in both *C* and assembly sources will be handled properly.

The – **c** option is used to specify the manner in which your assembler recognizes comments. The – **c** should be immediately followed by a character that is used to introduce a comment that extends to the end of line. If no comment delimiting character is specified, *mkid* will look for *C* style comments in assembly language text.

The – **a** option is used to extend *mkid*s notion of the alpha-numeric character class. It is quite common for assemblers to allow ., , or to appear in internal names. If this option is introduced with + *mkid* will ignore any names containing the class of characters specified here.

Finally, the – **v** stands for the *verbose* option. *Mkid* will echo each pathname as it is scanned and print statistics of the internal hash-table that it used while building the database.

SEE ALSO

lid(1).

NAME

mkstr – create an error message file by massaging C source

SYNOPSIS

mkstr [–] messagefile prefix file ...

DESCRIPTION

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

Mkstr will process each of the specified *files*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. A typical usage of *mkstr* would be

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file *mkstr* keys on the string 'error(" in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a new-line character; the null character terminates the message so it can be easily used when retrieved, the new-line character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file which can be used to retrieve the message, i.e.:

```
char    efilname[] = "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
{
        char buf[256];

        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
oops:
                        perror(efilname);
                        exit(1);
                }
        }
        if (lseek(efil, (long) a1, 0) ||read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

The optional – causes the error messages to be placed at the end of the specified message file for recompiling part of a large *mkstr*ed program.

SEE ALSO

lseek(2), xstr(1)

AUTHORS

William Joy and Charles Haley

NAME

mm, osdd, checkmm – print/check documents formatted with the MM macros

SYNOPSIS

mm [options] [files]

osdd [options] [files]

checkmm [files]

DESCRIPTION

Mm can be used to type out documents using *nroff*(1) and the MM text-formatting macro package. It has options to specify preprocessing by *tbl*(1) and/or *neqn* (see *eqn*(1)) and postprocessing by various terminal-oriented output filters. The proper pipelines and the required arguments and flags for *nroff*(1) and MM are generated, depending on the options selected.

Osdd is equivalent to the command **mm – mosd**. For more information about the OSDD adapter macro package, see *mosd*(5).

Options for *mm* are given below. Any other arguments or flags (e.g., **– rC3**) are passed to *nroff*(1) or to MM, as appropriate. Such options can occur in any order, but they must appear before the *files* arguments. If no arguments are given, *mm* prints a list of its options.

– T*term* Specifies the type of output terminal; for a list of recognized values for *term*, type **help term2**. If this option is *not* used, *mm* uses the value of the shell variable $TERM from the environment (see *profile*(4) and *environ*(5)) as the value of *term*, if $TERM is set; otherwise, *mm* uses **450** as the value of *term*. If several terminal types are specified, the last one takes precedence.

– 12 Indicates that the document is to be produced in 12-pitch. May be used when $TERM is set to one of **300**, **300s**, **450**, and **1620**. (The pitch switch on the DASI 300 and 300s terminals must be manually set to **12** if this option is used.)

– c Causes *mm* to invoke *col*(1); note that *col*(1) is invoked automatically by *mm* unless *term* is one of **300**, **300s**, **450**, **37**, **4000a**, **382**, **4014**, **tek**, **1620**, and **X**.

– e Causes *mm* to invoke *neqn*; also causes *neqn* to read the **/usr/pub/eqnchar** file (see *eqnchar*(5)).

– t Causes *mm* to invoke *tbl*(1).

– E Invokes the **– e** option of *nroff*(1).

– y Causes *mm* to use the non-compacted version of the macros (see *mm*(5)).

As an example (assuming that the shell variable $TERM is set in the environment to **450**), the two command lines below are equivalent:

mm – t – rC3 – 12 ghh*
tbl ghh* | nroff – cm – T450– 12 – h – rC3

Mm reads the standard input when **–** is specified instead of any filenames. (Mentioning other files together with **–** leads to disaster.) This option allows *mm* to be used as a filter, e.g.:

cat dws | mm –

Checkmm is a program for checking the contents of the named *files* for errors in the use of the Memorandum Macros, missing or unbalanced *neqn* delimiters, and **.EQ**/**.EN** pairs. Note: The user need not use the *checkeq* program (see *eqn*(1)). Appropriate messages are produced. The program skips all directories, and if no filename is given, standard input is read.

HINTS

1. *Mm* invokes *nroff*(1) with the **– h** flag. With this flag, *nroff*(1) assumes that the terminal has tabs set every 8 character positions.
2. Use the **– o***list* option of *nroff*(1) to specify ranges of pages to be output. Note, however, that *mm*, if invoked with one or more of the **– e**, **– t**, and **–** options, *together* with the **– o***list* option of *nroff*(1) may cause a harmless "broken pipe" diagnostic if the

last page of the document is not specified in *list*.

3. If you use the **– s** option of *nroff*(1) (to stop between pages of output), use line-feed (rather than return or new-line) to restart the output. The **– s** option of *nroff*(1) does not work with the **– c** option of *mm*, or if *mm* automatically invokes *col*(1) (see **– c** option above).
4. If you lie to *mm* about the kind of terminal its output is to be printed on, you get (often subtle) garbage; however, if you are redirecting output into a file, use the **– T37** option, and then use the appropriate terminal filter when you actually print that file.

SEE ALSO

col(1), cw(1), env(1), eqn(1), greek(1), mmt(1), nroff(1), tbl(1), profile(4), mm(5), mosd(5), term(5).
Document Processing Guide.

DIAGNOSTICS

mm **mm: no input file** means none of the arguments is a readable file and *mm* is not used as a filter.

checkmm **Cannot open** *filename* means *file* is unreadable. The remaining output of the program is diagnostic of the source file.

NAME

mmt, mvt – typeset documents, viewgraphs, and slides

SYNOPSIS

mmt [options] [files]

mvt [options] [files]

DESCRIPTION

These two commands are very similar to *mm*(1), except that they both typeset their input via *troff*(1), as opposed to formatting it via *nroff*(1); *mmt* uses the MM macro package, while *mvt* uses the Macro Package for Viewgraphs and Slides. These two commands have options to specify preprocessing by *tbl*(1) and/or *eqn*(1). The proper pipelines and the required arguments and flags for *troff*(1) and for the macro packages are generated, depending on the options selected.

Options are given below. Any other arguments or flags (e.g., **–rC3**) are passed to *troff*(1) or to the macro package, as appropriate. Such options can occur in any order, but they must appear before the *files* arguments. If no arguments are given, these commands print a list of their options.

- **–e** Causes these commands to invoke *eqn*(1); also causes *eqn* to read the **/usr/pub/eqnchar** file (see *eqnchar*(5)).
- **–t** Causes these commands to invoke *tbl*(1).
- **–Tst** Directs the output to the MH STARE facility.
- **–Tvp** Directs the output to a Versatec printer; this option is not available at all sites.
- **–T4014** Directs the output to a Tektronix 4014 terminal via the *tc*(1) filter.
- **–Ttek** Same as **–T4014**.
- **–a** Invokes the **–a** option of *troff*(1).
- **–y** Causes *mmt* to use the non-compacted version of the macros (see *mm*(5)). No effect for *mvt*.

These commands read the standard input when – is specified instead of any filenames.

Mvt is just a link to *mmt*.

HINT

Use the **–o***list* option of *troff*(1) to specify ranges of pages to be output. Note, however, that these commands, if invoked with one or more of the **–e**, **–t**, and – options, *together* with the **–o***list* option of *troff*(1) may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

SEE ALSO

env(1), eqn(1), mm(1), tbl(1), tc(1), troff(1), profile(4), environ(5), mm(5), mv(5).
Document Processing Guide.

DIAGNOSTICS

m[mv]t: no input file means none of the arguments is a readable file and the command is not used as a filter.

NAME

more, page – file perusal filter for crt viewing

SYNOPSIS

more [**–d**] [**–f**] [**–l**] [*–n*] [*+linenumber*] [*+/pattern*] [name ...]

page [**–d**] [**–f**] [**–l**] [*–n*] [*+linenumber*] [*+/pattern*] [name ...]

DESCRIPTION

More is a filter which allows examination of a continuous text one screenful at a time on a soft-copy terminal. It normally pauses after each screenful, printing --More-- at the bottom of the screen. If the user then types a carriage return, one more line is displayed. If the user hits a space, another screenful is displayed. Other possibilites are enumerated later.

The command line options are:

–n An integer which is the size (in lines) of the window which *more* will use instead of the default.

–d *More* will prompt the user with the message "Hit space to continue, Rubout to abort" at the end of each screenful. This is useful if *more* is being used as a filter in some setting, such as a class, where many users may be unsophisticated.

–f This causes *more* to count logical, rather than screen lines. That is, long lines are not folded. This option is recommended if *nroff* output is being piped through *ul*, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen postions, but which do not print when they are sent to the terminal as part of an escape sequence. Thus *more* may think that lines are longer than they actually are, and fold lines erroneously.

–l Do not treat ^L (form feed) specially. If this option is not given, *more* will pause after any line that contains a ^L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

+linenumber
Start up at *linenumber*.

+/pattern
Start up two lines before the line containing the regular expression *pattern*.

If the program is invoked as *page*, then the screen is cleared before each screenful is printed (but only if a full screenful is being printed), and $k - 1$ rather than $k - 2$ lines are printed in each screenful, where k is the number of lines the terminal can display.

More looks in the file */etc/termcap* to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

If *more* is reading from a file, rather than a pipe, then a percentage is displayed along with the --More-- prompt. This gives the fraction of the file (in characters, not lines) that has been read so far.

Other sequences which may be typed when *more* pauses, and their effects, are as follows (*i* is an optional integer argument, defaulting to 1) :

i<space>
display *i* more lines, (or another screenful if no argument is given)

^D display 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i*.

d same as ^D (control-D)

iz same as typing a space except that *i*, if present, becomes the new window size.

*i*s skip *i* lines and print a screenful of lines

*i*f skip *i* screenfuls and print a screenful of lines

q or Q Exit from *more*.

= Display the current line number.

v Start up the editor *vi* at the current line.

h Help command; give a description of all the *more* commands.

i/expr search for the *i*-th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

*i*n search for the *i*-th occurrence of the last regular expression entered.

' (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!command
invoke a shell with *command*. The characters '%' and '!' in "command" are replaced with the current file name and the previous shell command respectively. If there is no current file name, '%' is not expanded. The sequences "\%" and "\!" are replaced by "%" and "!" respectively.

i:n skip to the *i*-th next file given in the command line (skips to last file if n doesn't make sense)

i:p skip to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

:f display the current file name and line number.

:q or :Q
exit from *more* (same as q or Q).

. (dot) repeat the previous command.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command character itself is given, the user may hit the line kill character to cancel the numerical argument being formed. In addition, the user may hit the erase character to redisplay the --More--(xx%) message.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\). *More* will stop sending output, and will display the usual --More-- 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.

The terminal is set to *noecho* mode by this program so that the output can be continuous. What you type will thus not show on your terminal, except for the / and ! commands.

If the standard output is not a teletype, then *more* acts just like *cat*, except that a header is printed before each file (if there is more than one).

A sample usage of *more* in previewing *nroff* output would be

```
nroff -ms +2 doc.n |more
```

AUTHOR

Eric Shienbrood

FILES

/etc/termcap	Terminal data base
/usr/lib/more.help	Help file

SEE ALSO

script(1)

NAME

msgs – system messages and junk mail program

SYNOPSIS

msgs [– **fhlpq**] [number] [– number]

DESCRIPTION

Msgs is used to read system messages. These messages are sent by mailing to the login 'msgs' and should be short pieces of information which are suitable to be read once by most users of the system.

Msgs is normally invoked each time you login, by placing it in the file *.login (.profile* if you use */bin/sh*). It will then prompt you with the source and subject of each new message. If there is no subject line, the first few non-blank lines of the message will be displayed. If there is more to the message, you will be told how long it is and asked whether you wish to see the rest of the message. The possible responses are:

y type the rest of the message

RETURN

synonym for y.

n skip this message and go on to the next message.

– redisplay the last message.

q drops you out of *msgs;* the next time you run the program it will pick up where you left off.

s append the current message to the file "Messages" in the current directory; 's– ' will save the previously displayed message. A 's' or 's– ' may be followed by a space and a filename to receive the message replacing the default "Messages".

m or 'm– ' causes a copy of the specified message to be placed in a temporary mailbox and *mail*(1) to be invoked on that mailbox. Both 'm' and 's' accept a numeric argument in place of the '– '.

Msgs keeps track of the next message you will see by a number in the file *.msgsrc* in your home directory. In the directory */usr/msgs* it keeps a set of files whose names are the (sequential) numbers of the messages they represent. The file */usr/msgs/bounds* shows the low and high number of the messages in the directory so that *msgs* can quickly determine if there are no messages for you. If the contents of *bounds* is incorrect it can be fixed by removing it; *msgs* will make a new *bounds* file the next time it is run.

Options to msgs include:

– **f** which causes it not to say "No new messages.". This is useful in your *.login* file since this is often the case here.

– **q** Queries whether there are messages, printing "There are new messages." if there are. The command "msgs – q" is often used in login scripts.

– **h** causes *msgs* to print the first part of messages only.

– **l** option causes only locally originated messages to be reported.

num A message number can be given on the command line, causing *msgs* to start at the specified message rather than at the next message indicated by your *.msgsrc* file. Thus

msgs – h 1

prints the first part of all messages.

– *number*

will cause *msgs* to start *number* messages back from the one indicated by your *.msgsrc*

file, useful for reviews of recent messages.

-p causes long messages to be piped through *more*(1).

Within *msgs* you can also go to any specific message by typing its number when *msgs* requests input as to what to do.

FILES

/usr/msgs/*	database
~/.msgsrc	number of next message to be presented

AUTHORS

William Joy
David Wasley

SEE ALSO

mail(1), more(1)

BUGS

NAME

mt – magnetic tape manipulating program

SYNOPSIS

mt [**– f** *tapename*] *command* [*count*]

DESCRIPTION

Mt is used to give commands to a magnetic tape drive. If a tape name is not specified, the environment variable TAPE is used; if TAPE does not exist, *mt* uses the device */dev/rmt12*. Note that *tapename* must reference a raw (not block) tape device. By default *mt* performs the requested operation once. Operations may be performed multiple times by specifying *count*.

The available commands are listed below. Only as many characters as are required to uniquely identify a command need be specified.

eof, weof
Write *count* end-of-file marks at the current position on the tape.

fsf Forward space *count* files.

fsr Forward space *count* records.

bsf Back space *count* files.

bsr Back space *count* records.

rewind Rewind the tape (*Count* is ignored.)

offline, rewoffl
Rewind the tape and place the tape unit off-line (*Count* is ignored.)

status Print status information about the tape unit.

Mt returns a 0 exit status when the operation(s) were successful, 1 if the command was unrecognized, and 2 if an operation failed.

FILES

/dev/rmt* Raw magnetic tape interface

SEE ALSO

mtio(4), dd(1), ioctl(2), environ(7)

NAME

newaliases – rebuild the data base for the mail aliases file

SYNOPSIS

newaliases

DESCRIPTION

Newaliases rebuilds the random access data base for the mail aliases file */usr/lib/aliases*. It must be run each time */usr/lib/aliases* is changed in order for the change to take effect.

SEE ALSO

aliases(5), sendmail(8)

BUGS

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" yield results different from "– **l**60 – **e**15". Options are applied to all *files* on the command line.

– **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). If *tabspec* is – – , *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.

– **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).

– **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 filename
```

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.

– **e***n* Same as – **b***n* except that characters are truncated from the end of the line.

– **c***k* Change the prefix/append character to *k*. Default character for *k* is a space.

– **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 printed tab specification format line corresponds to the format specified in the last – **o** option. If no – **o** option is specified, the printed line contains the default specification of – **8**.

– **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 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, use the command:

newform – s – i – l – a – e filename

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).

EXIT CODES

0 – normal execution
1 – for any error

SEE ALSO

csplit(1), tabs(1), fspec(4).

BUGS

Newform normally only keeps track of physical characters; however, for the **– i** and **– o** options, *newform* keeps track of backspaces in order to line up tabs in the appropriate logical columns.

Newform does 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 **– o– –** or **– i– –**, the tab specification format line will be incorrect.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp [–] [group]

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login*(1). The same person remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

Newgrp without an argument changes the group identification to the group in the password file; in effect it changes the group identification back to the caller's original group.

An initial – flag causes the environment to be changed to the one that would be expected if the user actually logged in again.

A password is demanded if the group has a password and the user himself does not, or if the group has a password and the user is not listed in **/etc/group** as being a member of that group.

When most users log in, they are members of the group named **other**.

FILES

/etc/group
/etc/passwd

SEE ALSO

login(1), group(4).

BUGS

There is no convenient way to enter a password into **/etc/group**. Use of group passwords is not encouraged, because, by their very nature, they encourage poor security practices. Group passwords may disappear in the future.

NAME

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."

The **– a** option causes *news* to print all items, regardless of currency. In this case, the stored time is not changed.

The **– n** option causes *news* to report the names of the current items without printing their contents, and without changing the stored time.

The **– 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 one's **.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).

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).

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 [– htype] [– btype] [– ftype] [– vstart#] [– iincr] [– p] [– lnum] [– ssep] [– wwidth] [– nformat] [– ddelim] 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 filename. 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).

– **p** Do not restart numbering at logical page delimiters.

– **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**.

– **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 suppressed; **rz**, right justified, leading zeroes kept. Default *format* is **rn** (right justified).

– **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**.

– 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 file2

numbers files 1 and 2 starting at line number 10 with an increment of ten. The logical page delimiters are !+ .

SEE ALSO

pr(1).

NAME

nm - print name list of common object file

SYNOPSIS

nm [- a] [- d] [- o] [- x] [- h] [- v] [- n] [- e] [- f] [- u] [- V] [- T] filename(s)

DESCRIPTION

The *nm* command displays the symbol table of each common object file *filename*. *Filename* may be a relocatable or absolute common object file, or it may be an archive of relocatable or absolute common object files. *Nm* prints the following information for each symbol. Note that the object file must have been compiled with the **- g** option of the *cc*(1) command for there to be **Type**, **Size**, or **Line** information.

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Class Its storage class.

Tv If the symbol is accessed through a transfer vector, this field contains **tv**.

Type Its type and derived type. If the symbol is an instance of a structure or a union, the structure or union tag is given following the type (e.g., **struct-tag**). If the symbol is an array, the array dimensions are given following the type (e.g., **char**[**n**] [**m**]).

Size Its size in bytes, if available.

Line The source line number at which it is defined, if available.

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:

- d Print the value and size of a symbol in decimal instead of hexadecimal.

- o Print the value and size of a symbol in octal instead of hexadecimal.

- x Print the value and size of a symbol in hexadecimal (the default).

- 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 static and external symbols.

- f Produce full output. Redundant symbols (.test, .data, .bss), normally suppressed, are printed.

- u Print undefined symbols only.

- V Print the version of the *nm* command executing on the standard error output.

- T Truncate long names. By default, *nm* prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow, 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

/usr/tmp/nm??????

WARNINGS

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

DIAGNOSTICS

nm: name: cannot open	*Name* cannot be read.
nm: name: bad magic	*Name* is not an appropriate common object file.
nm: name: no symbols	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 redirected by the user, it is sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to $HOME/**nohup.out**.

SEE ALSO

nice(1), signal(2).

NAME

nroff - format text

SYNOPSIS

nroff [options] [files]

DESCRIPTION

Nroff formats text contained in *files* (standard input by default) for printing on typewriter-like devices and line printers. Its capabilities are described in the "NROFF/TROFF User's Manual" in the *Document Processing Guide*.

An argument consisting of a minus (-) is taken to be a file name corresponding to the standard input. The *options*, which may appear in any order, but must appear before the *files*, are:

- **-o***list* Print only pages whose page numbers appear in the *list* of numbers and ranges, separated by commas. A range *N*-*M* means pages *N* through *M*; an initial -*N* means from the beginning to page *N*; and a final *N*- means from *N* to the end. (See *BUGS* below.)
- **-n***N* Number first generated page *N*.
- **-s***N* Stop every *N* pages. *Nroff* halts *after* every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a line-feed or new-line (newlines do not work in pipelines, e.g., with *mm*(1)). This option does not work if the output of *nroff* is piped through *col*(1). When *nroff* halts between pages, an ASCII BEL is sent to the terminal.
- **-r***aN* Set register *a* (which must have a one-character name) to *N*.
- **-i** Read standard input after *files* are exhausted.
- **-q** Invoke the simultaneous input-output mode of the **.rd** request.
- **-z** Print only messages generated by **.tm** (terminal message) requests.
- **-m***name* Prepend to the input *files* the non-compacted (ASCII text) macro file **/usr/lib/tmac/tmac.***name*.
- **-c***name* Prepend to the input *files* the compacted macro files **/usr/lib/macros/cmp.**[**nt**]**.**[**dt**]**.***name* and **/usr/lib/macros/ucmp.**[**nt**]**.***name*.
- **-k***name* Compact the macros used in this invocation of *nroff*, placing the output in files [**dt**]**.***name* in the current directory (see the "NROFF/TROFF User's Manual" in the *Document Processing Guide* for details of compacting macro files).
- **-T***name* Prepare output for specified terminal. Known *name*s are **37** for the (default) TELETYPE Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **300s** for the DASI 300s, **300** for the DASI 300, **450** for the DASI 450, **lp** for a (generic) ASCII line printer, **382** for the DTC-382, **4000A** for the Trendata 4000A, **832** for the Anderson Jacobson 832, **X** for a (generic) EBCDIC printer, and **2631** for the Hewlett Packard 2631 line printer.
- **-e** Produce equally-spaced words in adjusted lines, using the full resolution of the particular terminal.
- **-h** Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.
- **-u***n* Set the emboldening factor (number of character overstrikes) for the third font position (bold) to *n*, or to zero if *n* is missing.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta$#	temporary file
/usr/lib/tmac/tmac.*	standard macro files and pointers
/usr/lib/macros/*	standard macro files
/usr/lib/term/*	terminal driving tables for *nroff*

SEE ALSO

"NROFF/TROFF User's Manual" in the *Document Processing Guide*

col(1), cw(1), eqn(1), greek(1), mm(1), tbl(1), troff(1), mm(5).

BUGS

Nroff believes in Eastern Standard Time: as a result, depending on the time of the year and on your local time zone, the date that *nroff* generates may be off by one day from your idea of what the date is.

When *nroff* is used with the *–olist* option inside a pipeline (e.g., with one or more of *cw*(1), *eqn*(1), and *tbl*(1)), it may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

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**, new-line=**\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.

An asterisk (*) appearing on a line by itself in the output indicates that the previous line is repeated until the next full line. This output commonly occurs when *od* is used on a file that contains multiple nulls.

SEE ALSO

dump(1).

NAME

pack, pcat, unpack – compress and expand files

SYNOPSIS

pack [–] 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*. If *pack* is successful, *name* is 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* 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 occurs if:

a. the file appears to be already packed;
b. the filename has more than 12 characters;
c. the file has links;
d. the file is a directory;
e. the file cannot be opened;
f. no disk storage blocks will be saved by packing;
g. a file called *name*.**z** already exists;
h. the **.z** file cannot be created; or
i. an I/O error occurred during processing.

The last segment of the filename 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. 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:

a. the filename (exclusive of the **.z**) has more than 12 characters;
b. the file cannot be opened; or

c. 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. a file with the "unpacked" name already exists; or
b. the unpacked file cannot be created.

SEE ALSO

cat(1)

NAME

passwd – change login password

SYNOPSIS

passwd name

DESCRIPTION

This command changes (or installs) a password associated with the login *name*.

The program prompts for the old password (if any) and then for the new one (twice). The caller must supply these. New passwords should be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. Only the first eight characters of the password are significant.

Only the owner of the name or the superuser may change a password; the owner must prove he knows the old password. Only the superuser can create a null password.

The password file is not changed if the new password is the same as the old password, or if the password has not "aged" sufficiently; see *passwd*(4).

FILES

/etc/passwd

SEE ALSO

login(1), crypt(3C), passwd(4).

NAME

paste – merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste file1 file2 ...
paste – dlist file1 file2 ...
paste – s [– dlist] file1 file2 ...

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). *Paste* is the counterpart of *cat*(1) which concatenates vertically, i.e., one file after the other. In the last form above, *paste* subsumes 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 filename.

The meanings of the options are:

– d Replace the *tab* character by one or more alternate characters specified in *list*. Without this option, the new-line characters of each but the last file (or last line in case of the **– s** option) are replaced by a *tab* character.

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 new-line character, not from the *list*. The list may contain the special escape sequences: **\n** (new-line), **\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 rather than one from each input file. Use *tab* for concatenation, unless a *list* is specified with the **– d** option. Regardless of the *list*, the last character of the file is forced to be a new-line.

– May be used in place of any filename, 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

grep(1), cut(1),
pr(1): **pr – t – m**... works similarly, but creates extra blanks, tabs and new-lines for a nice page layout.

DIAGNOSTICS

line too long Output lines are restricted to 511 characters.

too many files Except for the **– s** option, no more than 12 input files may be specified.

NAME

pma – post-mortem dump analyzer

SYNOPSIS

pma [-hcpkurlwbKB] [-f file] [-P pbr] [addr] [length]

DESCRIPTION

Pma is used to read the contents of a file generated by the SDU Monitor post-mortem dump utility **pmd.** *File* contains the dump image (the file name defaults to **pmdump** if not specified). If the dump was output to tape, the tape must be first copied to a disk file using **dd** with a blocksize of 8k. *Addr* specifies the starting address with the addressing unit defined by the options (defaults to zero if not specified). *Length* is the number of bytes from the starting address to dump (defaults to 256 if not specified).

h Display contents of dump file header.

c *addr* is in clicks.

p *addr* is NuBus physical address.

k *addr* is kernel virtual address.

u *addr* is user virtual address.

r byte reverse the display.

l display values as longs (4 bytes).

b display values as bytes.

K output binary core file image.

B output binary to stdout. This option is used with the structure filters in **libpma.** Each filter accepts binary from standard input, fills a structure, and displays the contents of the structure in a readable form. Each filter has the same name as the structure it displays.

f The dump file is specified by the next parameter.

P specify NuBus pbr address for virtual address decode.

defaults:

File is **pmdump.** *Addr* is absolute dump file offest in bytes if c, p, or k options are not specified. Values are displayed as words (2 bytes).

FILES

/usr/util/libpma directory of structure filters pmdump default dump file

SEE ALSO

dd(1), pmd(1) in SDU Operating System User Manual

NAME

pr – print files

SYNOPSIS

pr [options] [files]

DESCRIPTION

Pr prints the named files on the standard output. If *file* is – , or if no files are specified, the standard input is assumed. By default, the listing is separated into pages, each headed by the page number, a date and time, and the name of the file.

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 separation character.

If the standard output is associated with a terminal, error messages are withheld until *pr* has completed printing.

The *options* below may appear singly or be combined in any order:

+*k* Begin printing with page *k* (default is 1).

– *k* Produce *k*-column output (default is 1). The options **– e** and **– i** are assumed for multi-column output.

– a Print multi-column output across the page.

– m Merge and print all files simultaneously, one per column (overrides the – *k*, and **– a** options).

– d Double-space the output.

– 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 normal 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*k* Set the width of a line to *k* character positions (default is 72 for equal-width multi-column output, no limit otherwise).

– o*k* Offset each line by *k* character positions (default is 0). The number of character positions per line is the sum of the width and offset.

– l*k* Set the length of a page to *k* lines (default is 66).

– h Use the next argument as the header to be printed instead of the filename.

– 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 form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.

-r Print no diagnostic reports on failure to open files.

-t Print neither the 5-line identifying header nor the 5-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.

-sc Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

EXAMPLES

Print **file1** and **file2** as a double-spaced, three-column listing headed by "file list":

pr -3dh "file list" file1 file2

Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, 37, . . . :

pr -e9 -t <file1 >file2

FILES

/dev/tty* to suspend messages

SEE ALSO

cat(1).

NAME

printenv – print out the environment

SYNOPSIS

printenv [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a *name* is specified, only its value is printed.

If a *name* is specified and it is not defined in the environment, *printenv* returns exit status 1, else it returns status 0.

SEE ALSO

env(1), sh(1), environ(7), csh(1)

NAME

prof - display profile data

SYNOPSIS

prof [**- tcan**] [**- ox**] [**- g**] [**- z**] [**- h**] [**- s**] [**- m** mdata] [prog]

DESCRIPTION

Prof interprets the 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 the 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** for profiling data.

For the number of calls to a function to be tallied, the **- p** option of *cc*(1) must have been given when the file containing the function was compiled. This option to the *cc* command also arranges for the object file to include a special profiling start-up function that calls *monitor*(3C) at the beginning and end of execution. It is the call to *monitor* at the end of execution that causes the **mon.out** file to be written. Thus, only programs that call *exit*(2) or return from *main* cause the **mon.out** file to be produced.

FILES

mon.out for profile
a.out for namelist

SEE ALSO

cc(1), nm(1), exit(2), profil(2), monitor(3C).

BUGS

There is a limit of 600 functions that may have call counters established during program execution. If this limit is exceeded, other data is overwritten and the **mon.out** file is corrupted. The number of call counters used is reported automatically by the *prof* command whenever the number exceeds 250.

NAME

prs - print an SCCS file

SYNOPSIS

prs [- d[dataspec]] [- r[SID]] [- e] [- l] [- 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 filenames.

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 IDentification* (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.
- **- l** Requests information for all deltas created *later* than and including the delta designated via the **- r** keyletter.
- **- 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/new-line is specified by **\n**.

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 seq. #	"	nnnn	S
:DP:	Predecessor Delta seq. #	"	nnnn	S
:DI:	Seq. # 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
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/ warning flag	"	*yes* or *no*	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 filename	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. Only the – **a** keyletter argument can be used with the *special case*.

FILES

/tmp/pr?????

SEE ALSO

admin(1), delta(1), get(1), help(1), rmdel(1), sccsfile(4).
"Source Code Control System User's Guide" in the *User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ps – report process status

SYNOPSIS

ps [options]

DESCRIPTION

Ps prints certain information about active processes. Without *options*, information is printed about processes associated with the current terminal. Otherwise, the displayed information is controlled by the following *options*:

- **– e** Print information about all processes.
- **– d** Print information about all processes, except process group leaders.
- **– a** Print information about all processes, except process group leaders and processes not associated with a terminal.
- **– f** Generate a *full* listing. Normally, a short listing containing only process ID, terminal ("tty") identifier, cumulative execution time, and the command name is printed. See below for meaning of columns in a full listing.
- **– l** Generate a *long* listing. See below.
- **– c** *corefile* Use the file *corefile* in place of **/dev/mem**.
- **– s** *swapdev* Use the file *swapdev* in place of **/dev/swap**. This is useful when examining a *corefile*; a *swapdev* of **/dev/null** causes the user block to be zeroed out.
- **– n** *namelist* The argument is taken as the name of an alternate *namelist* (**/unix** is the default).
- **– t** *tlist* Restrict listing to data about the processes associated with the terminals given in *tlist*, where *tlist* can be in one of two forms: a list of terminal identifiers separated by commas, or a list of terminal identifiers enclosed in double quotes and separated by a comma and/or one or more spaces.
- **– p** *plist* Restrict listing to data about processes whose process ID numbers are given in *plist*, where *plist* is in the same format as *tlist*.
- **– u** *ulist* Restrict listing to data about processes whose user ID numbers or login names are given in *ulist*, where *ulist* is in the same format as *tlist*. In the listing, the numerical user ID is printed unless the **– f** option is used, in which case the login name is printed.
- **– g** *glist* Restrict listing to data about processes whose process groups are given in *glist*, where *glist* is a list of process group leaders and is in the same format as *tlist*.

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

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

01	in core;
02	system process;
04	locked in core (e.g., for physical I/O);
10	being swapped;
20	being traced by another process;
40	another tracing flag.

S (l) The state of the process:

0	non-existent;
S	sleeping;
W	waiting;
R	running;
I	intermediate;
Z	terminated;

T stopped;
X growing.

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; it is possible to kill a process if you know the **PID**.
PPID	(f,l)	The process ID of the parent process.
C	(f,l)	Processor utilization for scheduling.
STIME	(f)	Starting time of the process.
PRI	(l)	The priority of the process; higher numbers mean lower priority.
NI	(l)	Nice value; used in priority computation.
RSSZ	(l)	The number of data and stack pages currently in core.
SZ	(l)	The number of pages in the process's data and stack segments.
WCHAN	(l)	The event for which the process is waiting or sleeping; if blank, the process is running.
TTY	(all)	The controlling terminal for the process.
TIME	(all)	The cumulative execution time for the process.
CMD	(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**>.

Under the – **f** option, *ps* tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the – **f** option, is printed in square brackets.

FILES

/unix	system namelist.
/dev/mem	memory.
/dev/swap	the default swap device.
/etc/passwd	supplies UID information.
/etc/ps_data	internal data structure.
/dev	searched to find terminal ("tty") names.

SEE ALSO

kill(1), nice(1).

BUGS

Things can change while *ps* is running; the picture it gives is only a close approximation to reality. Some data printed for defunct processes are irrelevant.

NAME

ptx – permuted index

SYNOPSIS

ptx [options] [input [output]]

DESCRIPTION

Ptx generates the file *output* that can be processed with a text formatter to produce a permuted index of file *input* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of each line. *Ptx* output is in the form:

.xx "tail" "before keyword" "keyword and after" "head"

where **.xx** is assumed to be an *nroff* or *troff*(1) macro provided by the user, or provided by the *mptx*(5) macro package. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed. *Tail* and *head*, at least one of which is always the empty string, are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line.

The following *options* can be applied:

– **f** Fold upper and lower case letters for sorting.

– **t** Prepare the output for the phototypesetter.

– **w** *n* Use the next argument, *n*, as the length of the output line. The default line length is 72 characters for *nroff* and 100 for *troff*.

– **g** *n* Use the next argument, *n*, as the number of characters that *ptx* reserves in its calculations for each gap among the four parts of the line as finally printed. The default gap is 3.

– **o** *only* Use as keywords only the words given in the *only* file.

– **i** *ignore* Do not use as keywords any words given in the *ignore* file. If the – **i** and – **o** options are missing, use **/usr/lib/eign** as the *ignore* file.

– **b** *break* Use the characters in the *break* file to separate words. Tab, new-line, and space characters are always used as break characters.

– **r** Take any leading non-blank characters of each input line to be a reference identifier (e.g., a page or chapter reference), separate from the text of the line. Attach the reference identifier as a 5th field on each output line.

The index for this manual was generated using *ptx*.

FILES

/bin/sort
/usr/lib/eign
/usr/lib/tmac/tmac.ptx

SEE ALSO

nroff(1), troff(1), mm(5), mptx(5).

BUGS

Line length counts do not account for overstriking or proportional spacing.
Lines that contain tildes (~) are botched, because *ptx* uses that character internally.

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

SEE ALSO

cd(1).

DIAGNOSTICS

Cannot open .. and **Read error in ..** indicate possible file system trouble. These messages should be referred to a system programming counselor.

NAME

ratfor – rational Fortran dialect

SYNOPSIS

ratfor [options] [files]

DESCRIPTION

Ratfor converts a rational dialect of Fortran into ordinary irrational Fortran. *Ratfor* provides control flow constructs essentially identical to those in C:

statement grouping:

{ statement; statement; statement }

decision-making:

if (condition) statement [**else** statement]
switch (integer value) {
    case integer: statement
    ...
    [**default**:] statement
}

loops:

while (condition) statement
for (expression; condition; expression) statement
do limits statement
repeat statement [**until** (condition)]
break
next

and some syntactic sugar to make programs easier to read and write:

free form input:

multiple statements/line; automatic continuation

comments:

this is a comment.

translation of relationals:

>, >=, etc., become **.GT.**, **.GE.**, etc.

return expression to caller from function:

return (expression)

define:

define *name replacement*

include:

include *file*

The option – **h** causes quoted strings to be turned into **27H** constructs. The – **C** option copies comments to the output and attempts to format it neatly. Normally, continuation lines are marked with a & in column 1; the option – **6x** makes the continuation character **x** and places it in column 6.

Ratfor is best used with *f77*(1).

SEE ALSO

efl(1), f77(1).
B. W. Kernighan and P. J. Plauger, "Software Tools", Addison-Wesley, 1976.
"Fortran" in the *Programming Guide*.

NAME

regcmp – regular expression compile

SYNOPSIS

regcmp [–] files

DESCRIPTION

Regcmp, in most cases, precludes the need for calling *regcmp*(3X) from C programs. This saves 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 is 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* into C programs, or *file*.**c** files may be compiled and later loaded. In the C program which uses the *regcmp* output, *regex*(*abc*,*line*) applies 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)

applies the regular expression named *telno* to *line*.

SEE ALSO

regcmp(3X).

NAME

rm, rmdir – remove files or directories

SYNOPSIS

rm [**– fri**] file ...

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry is the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with **y** the file is deleted; otherwise, the file remains. No questions are asked when the **– f** option is given or if the standard input is not a terminal.

If a designated file is a directory, an error comment is printed unless the optional argument **– r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory, and the directory itself.

If the **– i** (interactive) option is in effect, *rm* asks whether to delete each file, and, under **– r**, whether to examine each directory.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

unlink(2).

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file **..** merely to avoid the consequences of inadvertently doing something like:

rm – r .*

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 *SID* 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 *SID* must *not* appear in any entry of the *p-file*).

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 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; non-SCCS files and unreadable files are silently ignored.

The exact permissions necessary to remove a delta are documented in the "Source Code Control System User's Guide". 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), help(1), prs(1), sccsfile(4).
"Source Code Control System User's Guide" in the *User's Guide*.

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).
"Source Code Control System User's Guide" in the *User's Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sar – system activity reporter

SYNOPSIS

sar [– **ubdycwaqvmA**] [– **o** file] t [n]

sar [– **ubdycwaqvmA**] [– **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. If the – **o** option is specified, *sar* 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 – **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 the following options:

– **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.

– **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, e. g., 1 – bread/lread;
pread/s, pwrit/s – transfers via raw (physical) device mechanism.

– **d** Report activity for each block device, e. g., disk or tape drive:
%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.

– **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 (including initial loading of some programs) and swapouts;
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 text, process, inode and file tables:
text-sz, proc-sz, inod-sz, file-sz – entries/size for each table, evaluated once at sampling point;
text-ov, proc-ov, inod-ov, file-ov – overflows occurring between sampling points.

– m Report message and semaphore activities:
msg/s, sema/s – primitives per second.

– A Report all data. Equivalent to **– udqbwcayvm.**

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) in the *Administrator's Manual*.

NAME

scat – concatenate and print files on synchronous printer

SYNOPSIS

scat [**–u**] [**–s**] file ...

DESCRIPTION

Scat reads each *file* in sequence and writes it on the standard output, which is assumed to be a synchronous printer device. Thus:

scat file > /dev/sp0

prints the file, and:

scat file1 file2 > /dev/sp0

concatenates *file1* and *file2* and places the result on the printer.

If no input file is given, or if the argument – is encountered, *scat* reads from the standard input file. Output is buffered in 512-byte blocks unless the **–u** option is specified. The **–s** option makes *scat* silent about non-existent files.

SEE ALSO

cp(1), pr(1), stty(1).

WARNINGS

Scat uses synchronous printers in line mode with the wrap around option enabled. This means that the maximum line length is 79 characters; longer lines are wrapped back to the beginning of the next line each time the end of a printer line is reached.

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff – rSID1 – rSID2 [– p] [– sn] 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

bdiff(1), get(1), help(1), pr(1).
"Source Code Control System" in the *User's Guide*.

DIAGNOSTICS

file: **No differences** means the two versions are the same.

Use *help*(1) for explanations.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [– **w**] [– **W**] [objfil [corfil [directory]]]

DESCRIPTION

Sdb is a symbolic debugger which can be used with C and F77 programs. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

Objfil is normally an executable program file which has been compiled with the – **g** (debug) option; if it has not been compiled with the – **g** option, or if it is not an executable file, the symbolic capabilities of *sdb* are 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* forces *sdb* to ignore any core image file. Source files used in constructing *objfil* must be in *directory* to be located.

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 or F77. 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.

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* interprets a structure as a set of variables; thus, it displays 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 filename 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*). 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 printed numerically only.

The length specifiers are only effective with the formats **c**, **d**, **u**, **o** and **x**. 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, then 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, 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 that 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, filename, or directory is given, the current procedure name and filename 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 elided.

?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 elided.

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, it specifies the number of breakpoints to be ignored. **C** continues with the signal that caused the program to stop reactivated and **c** ignores it. If a linenumber 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. **I** steps with the signal that caused the program to stop reactivated and **i** ignores it.

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**.

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, 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**, 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).

new-line

If the previous command printed a source line, advance the current line by one line and print the new current line. If the previous command displayed a memory location, display the next memory location.

control-D
Scroll. Print the next 10 lines of instructions, source, or data, depending on which was printed last.

< *filename*
Read commands from *filename* until the end of file is reached, 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, (*b1, e1, f1*), 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.

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.
Y Toggle debug output.

FILES

a.out
core

SEE ALSO

cc(1), f77(1), sh(1), a.out(4), core(4).
Programming Guide.

WARNINGS

Data stored in text sections are indistinguishable from functions.

Line number information in optimized functions is unreliable, and some information may be missing.

BUGS

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.

The default type for printing F77 parameters is incorrect. Their address is printed instead of their value.

Tracebacks containing F77 subprograms with multiple entry points may print too many arguments in the wrong order, but their values are correct.

The range of an F77 array subscript is assumed to be *1* to *n*, where *n* is the dimension corresponding to that subscript. This is only significant when the user omits a subscript, or uses * to indicate the full range. There is no problem in general with arrays having subscripts whose lower bounds are not 1.

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 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** sfile] [files]

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** option, 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 new-line *embedded* in the pattern space.

A period **.** matches any character except the *terminal* new-line 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 new-line. 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 new-line. 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-printing characters are spelled in two-digit ASCII 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 new-line. (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 new-line 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:

- **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.

SEE ALSO

awk(1), ed(1), grep(1).

"Document Preparation, Stream Editor" in the *Document Processing Guide*.

NAME

sfnt – select loaded font

SYNOPSIS

sfnt [fontnum]

DESCRIPTION

Sfnt selects the current font in use to be *fontnum*. The font must have already been loaded into the window using **lfnt**. *Fontnum* must be in the range 0 to 7. If *fontnum* is not specified, it defaults to 7 which is the default font for the window.

SEE ALSO

lfnt(1) lsfnt(1) cfnt(1)

NAME

sh, rsh – shell, the standard/restricted command programming language

SYNOPSIS

sh [– **ceiknrstuvx**] [args]
rsh [– **ceiknrstuvx**] [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* below for the meaning of arguments to the shell.

Commands.

A *simple-command* is a sequence of non-blank *words* separated by *blanks* (a *blank* is a tab or a space). 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 **|** (or, for historical compatibility, by ^). The standard output of each command except the last one 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.

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 new-lines 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 filename generation (see *Filename Generation* below).

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 simply executed.

The following words are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { }

Comments.

A word beginning with # causes that word and all the following characters up to a new-line to be ignored.

Command Substitution.

The standard output from a command enclosed in a pair of grave accents (`` ` ` ``) may be used as part or all of a word; trailing new-lines are removed.

Parameter Substitution.

The character $ is used to introduce substitutable *parameters*. Positional parameters may be assigned values by **set**. Variables may be set by writing:

name=*value* [*name*=*value*] ...

Pattern-matching is not performed on *value*.

${*parameter*}

A *parameter* is a sequence of letters, digits, or underscores (a *name*), a digit, or any of the characters *, @, #, ?, -, $, and !. 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. A *name* must begin with a letter or underscore. If *parameter* is a digit then it is a positional parameter. If *parameter* is * or @, then 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 then substituted. Positional parameters may not be assigned 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 structures, *word* is not evaluated unless it is to be used as the substituted string; 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 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* below). The user may not change PATH if executing under *rsh*.

CDPATH The search path for the *cd* command.

MAIL If this variable is set to the name of a mail file, then the shell informs the user of the arrival of mail in the specified file.

PS1 Primary prompt string, by default "$ ".

PS2 Secondary prompt string, by default "> ".

IFS Internal field separators, normally **space**, **tab**, and **new-line**.

The shell gives default values to PATH, PS1, PS2, and IFS, while HOME and MAIL are not set at all by the shell (although HOME *is* 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.

Filename Generation.

Following substitution, each command *word* is scanned for the characters *, ?, and [. If one of these characters appears then the word is regarded as a *pattern*. The word is replaced with alphabetically sorted filenames that match the pattern. If no filename is found that matches the pattern, then the word is left unchanged. The character . at the start of a filename or immediately following a /, as well as the character / itself, must be matched explicitly.

* Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening "[" is a "!" then any character not enclosed is matched.

Quoting.

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

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

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a \. The pair **\new-line** is ignored. All characters enclosed between a pair of single quote marks (''), except a single quote, are quoted. Inside double quote marks (""), parameter and command substitution occurs and \ quotes the characters \, `, ", and $. "$*" is equivalent to "$1 $2 ...", whereas "$@" is equivalent to "$1" "$2" ....

Prompting.

When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a new-line is typed and further input is needed to complete a command, then 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; substitution occurs before *word* or *digit* is used:

<word	Use file *word* as standard input (file descriptor 0).
>word	Use file *word* as standard output (file descriptor 1). If the file does not exist, it is created; otherwise, it is truncated to zero length.
>>word	Use file *word* as standard output. If the file exists, output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<<[-]word	The shell input is read up to a line that is the same as *word*, or to an end-of-file. The resulting document becomes the standard input. If any character of *word* is quoted, then no interpretation is placed upon the characters of the document; otherwise, parameter substitution and command substitution occur, (unescaped) **\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)). The standard output can be duplicated similarly, using **>** in place of **<**.
<&-	The standard input is closed. The standard output can be closed similarly, using **>** in place of **<**.

If one of the above is preceded by a digit, the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

creates file descriptor 2 that is a duplicate of file descriptor 1.

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.

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. Executed commands inherit the same environment. If the user modifies the values of these parameters or creates new ones, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment. The environment seen by an executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, 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 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* keyword 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
```

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 below).

Execution.

Each time a command is executed, the above substitutions are carried out. Except for the *Special Commands* listed below, 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 path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a /, the search path is not used; such commands are not executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a sub-shell.

Special Commands.

The following commands are executed in the shell process and, except as specified, no input/output redirection is permitted for such commands:

: 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 then break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified then resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter HOME is the default *arg*. The shell parameter CDPATH defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a / then 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*.

eval [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*]
Causes a shell to exit with the exit status specified by *n*. If *n* is omitted, the exit status is that of the last command executed (an end-of-file also causes the shell to exit.)

export [*name* ...]
The given *names* are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, a list of all names that are exported in this shell is printed.

newgrp [*arg* ...]
Equivalent to **exec newgrp** *arg* ....

read [*name* ...]
One line is read from the standard input and 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 return code is 0 unless an end-of-file is encountered.

readonly [*name* ...]

The given *names* are marked *readonly* and the values of the these *names* may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed.

set [– – **ekntuvx** [*arg* ...]]

– **e** Exit immediately if a command exits with a non-zero exit status.

– **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*] ...

arg is a command 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 then all trap(s) *n* 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 *n* is 0 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.

ulimit [– **fp**] [*n*]

imposes a size limit of *n*

– **f** imposes a size limit of *n* blocks on files written by child processes (files of any size may be read). With no argument, the current limit is printed.

– **p** changes the pipe size to *n* (UNIX System/RT only).

If no option is given, – **f** is assumed.

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.

wait [*n*]

Wait for the specified process and report its termination status. If *n* is not given, all currently active child processes are waited for and the return code is zero.

Invocation.

If the shell is invoked through *exec*(2) and the first character of argument zero is – , commands are initially read from **/etc/profile** and then 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 then 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 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, the 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.

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** 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 *rsh*. 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 then 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), env(1), login(1), newgrp(1), test(1), umask(1), dup(2), exec(2), fork(2), pipe(2), signal(2), ulimit(2), umask(2), wait(2), a.out(4), profile(4), environ(5).
"An Introduction to Shell" in the *Programming Guide*.

BUGS

The command **readonly** (without arguments) produces the same output as the command **export**.

If << is used to provide standard input to an asynchronous process invoked by &, the shell gets mixed up about naming the input document; a garbage file **/tmp/sh*** is created and the shell complains about not being able to find that file by another name.

NAME

size – print section sizes of common object files

SYNOPSIS

size [– **o**] [– **d**] [– **V**] filename(s)

DESCRIPTION

The *size* command produces section size information for each section in the common object files. The name of the section is printed followed by its size in bytes, physical address, and virtual address.

Numbers are printed in hexadecimal unless either the – **o** or the – **d** option is used, in which case they are printed in octal or in decimal, respectively.

The – **V** flag supplies the version information on the *size* command.

SEE ALSO

as(1), cc(1), ld(1).

DIAGNOSTICS

size: name: cannot open	*Name* cannot be read.
size: name: bad magic	*Name* is not an 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 under specified conditions, as in:

```
while true
do
        command
        sleep 37
done
```

SEE ALSO

alarm(2), sleep(3C).

BUGS

Time must be less than 65536 seconds.

NAME

sno – SNOBOL interpreter

SYNOPSIS

sno [files]

DESCRIPTION

Sno is a SNOBOL compiler and interpreter (with slight differences). *Sno* obtains input from the concatenation of the named *files* and the standard input. All input through a statement containing the label **end** is considered program and is compiled. The rest is available to **syspit**.

Sno differs from SNOBOL in the following ways:

There are no unanchored searches. To get the same effect:

```
a ** b          unanchored search for b.
a *x* b = x c   unanchored assignment
```

There is no back referencing.

```
x = "abc"
a *x* x         is an unanchored search for abc.
```

Function declaration is done at compile time by the use of the (non-unique) label **define**. Execution of a function call begins at the statement following the **define**. Functions cannot be defined at run time, and the use of the name **define** is preempted. There is no provision for automatic variables other than parameters. Examples:

```
define f( )
define f(a, b, c)
```

All labels except **define** (even **end**) must have a non-empty statement.

Labels, functions and variables must all have distinct names. In particular, the non-empty statement on **end** cannot merely name a label.

If **start** is a label in the program, program execution starts there. If not, execution begins with the first executable statement; **define** is not an executable statement.

There are no built-in functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators / and * must be set off by spaces.

The right side of assignments must be non-empty.

Either ' or " may be used for literal quotes.

The pseudo-variable **sysppt** is not available.

SEE ALSO

awk(1).

SNOBOL, A String Manipulation Language, by D. J. Farber, R. E. Griswold, and I. P. Polonsky, *JACM* **11** (1964), pp. 21-30.

NAME

sort – sort and/or merge files

SYNOPSIS

sort [– **cmubdfinrtx**] [+pos1 [– pos2]] ... [– **o** output] [names]

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name – means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d Sort in dictionary order; i.e., only letters, digits, and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.

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. Option **n** implies option **b**.

r Reverse the sense of comparisons.

t*x* Tab character separating fields is *x*.

The notation +*pos1* – *pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr**, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect, *n* is counted from the first non-blank in the field; **b** is attached independently to *pos2*. A missing .*n* means .0; a missing – *pos2* means the end of the line. Under the – **t***x* option, fields are strings separated by *x*; otherwise fields are non-empty non-blank strings separated by blanks.

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.

These option arguments are also understood:

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 Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

o The next argument 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.

EXAMPLES

Print in alphabetical order all the unique spellings in a list of words (capitalized words differ from uncapitalized):

sort – u +0f +0 list

Print the password file (*passwd*(4)) sorted by user ID (the third colon-separated field):

sort – t: +2n /etc/passwd

Print the first instance of each month in an already sorted file of (month-day) entries (the options **– um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

sort – um +0 – 1 dates

FILES

/usr/tmp/stm???

SEE ALSO

comm(1), join(1), uniq(1).

DIAGNOSTICS

Comments and exits with non-zero status for various trouble conditions and for disorder discovered under option **– c**.

BUGS

Very long lines are silently truncated.

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*, etc., 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* follows 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), eqn(1), sed(1), sort(1), tbl(1), tee(1), troff(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*.

The British spelling feature was done by an American.

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, the standard input file is used.

SEE ALSO

bfs(1), csplit(1).

NAME

strings – find the printable strings in a object, or other binary, file

SYNOPSIS

strings [–] [– **o**] [– *number*] file ...

DESCRIPTION

Strings looks for ascii strings in a binary file. A string is any sequence of 4 or more printing characters ending with a newline or a null. Unless the – flag is given, *strings* only looks in the initialized data space of object files. If the – **o** flag is given, then each string is preceded by its offset in the file (in octal). If the – *number* flag is given then number is used as the minimum string length rather than 4.

Strings is useful for identifying random object files and many other things.

SEE ALSO

od(1)

BUGS

The algorithm for identifying strings is extremely primitive

NAME

strip - strip symbol and line number information from an object file

SYNOPSIS

strip [- l] [- x] [- r] [- s] [- V] filename(s)

DESCRIPTION

The *strip* command strips the symbol table and line number information from object files, including archives. When *strip* has been performed, no symbolic debugging access is 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 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.
- **- r** Reset the relocation indexes into the symbol table.
- **- s** Reset the line number indexes into the symbol table (do not remove). Reset the relocation indexes into the symbol table.
- **- 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* complains and terminates without stripping *filename* unless the **- r** flag is used.

If the *strip* command is executed on a common archive file (see *ar*(4)) the archive symbol table is 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*(1) instructs the user with appropriate warning messages when this situation arises.

The purpose of this command is to reduce the file storage overhead taken by the object file.

FILES

/usr/tmp/str??????

SEE ALSO

as(1), cc(1), ld(1), ar(4), a.out(4).

DIAGNOSTICS

strip: name: cannot open *name* cannot be read.

strip: name: bad magic *name* is not an object file.

strip: name: relocation entries present; cannot strip
name contains relocation entries and the **- r** flag was not used; therefore, 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; with the **– a** option, it reports all of the option settings; with the **– g** option, it reports current settings in a form that can be used as an argument to another *stty* command. Detailed information about the modes listed in the first five groups below is provided in *termio*(7). 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.

50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb
Set terminal baud rate to the number given, if possible. (All speeds are not supported by all hardware interfaces.)

hupcl (**– hupcl**) hang up (do not hang up) a DATA-PHONE data set 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.

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) lower-case alphabetics to upper case on output.

onlcr (**– onlcr**) map (do not map) NL to CR-NL on output.

ocrnl (**– ocrnl**) map (do not map) CR to NL on output.

onocr (**– onocr**) do not (do) output CRs at column zero.

onlret (**– onlret**) on the terminal NL performs (does not perform) the CR function.

ofill (**– ofill**) use fill characters (use timing) for delays.

ofdel (– **ofdel**)	fill characters are DELs (NULs).
cr0 cr1 cr2 cr3	select style of delay for carriage returns (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 and QUIT.
icanon (– **icanon**)	enable (disable) canonical input (ERASE and KILL processing).
xcase (– **xcase**)	canonical (unprocessed) upper/lower-case presentation.
echo (– **echo**)	echo back (do not echo back) every character typed.
echoe (– **echoe**)	echo (do not echo) ERASE character as a backspace-space-backspace string. Note: 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 or QUIT.
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**, **eof**, **eol**, **min** ,or time (**min** and **time** are used with – **icanon**; see *termio*(7)). If *c* is preceded by an (escaped from the shell) caret (^), then the value used is the corresponding CTRL character (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, 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).

termio(7) in the *Adminstrator's Manual*.

NAME

su – become superuser or another user

SYNOPSIS

su [–] [name [arg ...]]

DESCRIPTION

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

To use *su*, the appropriate password must be supplied (unless one is already superuser). If the password is correct, *su* executes a new shell with the user ID set to that of the specified user. To restore normal user ID privileges, type an **EOF** to the new shell.

Any additional arguments are passed to the shell, permitting the superuser to run shell procedures with restricted privileges (an *arg* of the form **– c** *string* executes *string* via the shell). When additional arguments are passed, **/bin/sh** is always used. When no additional arguments are passed, *su* uses the shell specified in the password file.

An initial – flag causes the environment to be changed to the one that would be expected if the user actually logged in again. This is done by invoking the shell with an *arg0* of **– su**, causing the **.profile** in the home directory of the new user ID to be executed. Otherwise, the environment is passed along with the possible exception of $PATH, which is set to **/bin:/etc:/usr/bin** for root. Note that the **.profile** can check *arg0* for **– sh** or **– su** to determine how it was invoked.

FILES

/etc/passwd	system password file
$HOME/.profile	user's profile

SEE ALSO

env(1), login(1), sh(1), environ(5).

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 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

supdup – user interface to the SUPDUP protocol

SYNOPSIS

supdup [host]

DESCRIPTION

Supdup is used to communicate with another host using the SUPDUP protocol. If *supdup* is invoked without an argument, it enters command mode, indicated by its prompt (supdup>). In this mode, it accepts and executes the commands listed below. If it is invoked with an argument, it performs an *open* command (see below) with that argument.

Once a connection has been opened, *supdup* enters input mode. In this mode, text typed is sent to the remote host. To issue *supdup* commands when in input mode, precede them with the *supdup* escape character (initially ^@). When in command mode, the normal terminal editing conventions are available.

The following commands are available. Only enough of each command to uniquely identify it need be typed.

open *host*
Open a connection to the named host and contact the SUPDUP server on that host. The host specification may be either a host name (see *hosts*(5)) or an Internet address specified in the dot notation.

bye Close a SUPDUP session and return to command mode.

quit Close any open SUPDUP session and exit *supdup*.

escape [*escape-char*]
Set the *supdup* escape character. Control characters may be specified as ^ followed by a single letter; e.g. control-X is ^X.

status Show the current status of *supdup*. This includes the peer one is connected to, as well as the state of debugging.

help [*command*]
Get help. With no arguments, *supdup* prints a help summary. If a command is specified, *supdup* will print the help information available about the command only.

? A synonym for help.

connect
A synonym for open.

disconnect
A synonym for bye.

NAME

sync – update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to insure file system integrity. It flushes all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point are saved. See *sync*(2) for details.

SEE ALSO

sync(2).

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [tabspec] [+mn] [– Ttype]

DESCRIPTION

Tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user must of course be logged in on a terminal with remotely-settable hardware tabs.

Users of GE TermiNet terminals should be aware that they behave in a different way than most other terminals for some tab settings: the first number in a list of tab settings becomes the *left margin* on a TermiNet terminal. Thus, any list of tab numbers whose first element is other than 1 causes a margin to be left on a TermiNet, but not on other terminals. A tab list beginning with 1 causes the same effect regardless of terminal type. It is possible to set a left margin on some other terminals, although in a different way (see below).

Four types of tab specification are accepted for *tabspec*: "canned," repetitive, arbitrary, and file. If no *tabspec* is given, the default value is **– 8**, i.e., "standard" System V 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* Gives the name of one of a set of "canned" tabs. The legal codes and their meanings are as follows:

– 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:

<: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:

<: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

In addition to these "canned" formats, three other types exist:

– *n* A repetitive specification requests tabs at columns 1+ *n*, 1+ 2**n*, etc. Note that such a setting leaves a left margin of *n* columns on TermiNet terminals *only*. 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. It is required for use with the *nroff* **– h** option for high-speed output. 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 tab lists 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. 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 may be used also; 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* searches for the $TERM value in the *environment* (see *environ*(5)). If no *type* can be found, *tabs* tries a sequence that works 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.

DIAGNOSTICS

illegal tabs	arbitrary tabs are ordered incorrectly.
illegal increment	a zero or missing increment is found in an arbitrary specification.
unknown tab code	a "canned" code cannot be found.
can't open	**– –** *file* option was used, and file can't be opened.
file indirection	**– –** *file* option was used and the specification in that file points to yet another file. Indirection of this form is not permitted.

SEE ALSO

nroff(1), environ(5), term(5).

BUGS

There is no consistency among different terminals regarding ways of clearing tabs and setting the left margin.

It is generally impossible to usefully change the left margin without also setting tabs.

Tabs clears only 20 tabs (on terminals requiring a long sequence), but can set 40 tabs.

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 does not terminate after the line of the input file has been copied, but enters 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

prints 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

prints 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(1).

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [± number[**lbc**][**fr**]] [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. *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.

Specifying **r** causes tail to print lines from the end of the file in reverse order. The default for **r** is to print the entire file this way. Specifying **f** causes *tail* to not quit at end of file, but rather wait and try to read repeatedly in hopes that the file will grow.

SEE ALSO

dd(1)

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length.

Various kinds of anomalous behavior may happen with character special files.

NAME

tar – tape file archiver

SYNOPSIS

tar [key] [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 at most one function letter and possibly one or more function modifiers. 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 The named *files* are written on the end of the tape. The **c** function implies this function.

x 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. 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 The names of the specified files are listed each time that they occur on the tape. If no *files* argument is given, all the names on the tape are listed.

u The named *files* are added to the tape if they are not already there, or have been modified since last written on that tape.

c A new tape is created; writing begins at the beginning of the tape, instead of after the last file. This command implies the **r** function.

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

0,...,7 This modifier selects the drive on which the tape is mounted. The default is **1**.

v 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 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".

f causes *tar* to use the next argument as the name of the archive instead of **/dev/mt?**. 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 causes *tar* to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 20. This option should be used only with raw magnetic tape archives (see **f** above). The block size is determined automatically when reading tapes (key letters **x** and **t**).

l 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 tells *tar* to not restore the modification times. The modification time of the file is the time of extraction.

FILES

/dev/mt?
/tmp/tar*

DIAGNOSTICS

Complains about bad key characters and tape read/write errors.
Complains 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 **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 filename length is 100 characters.

NAME

tbl – format tables for nroff or troff

SYNOPSIS

tbl [– TX] [files]

DESCRIPTION

Tbl is a preprocessor that formats tables for *nroff*(1) or *troff*(1). The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables and are reformatted by *tbl*. The .TS and .TE command lines are not altered by *tbl*.

.TS is followed by global options. The available global options are:

center	center the table (default is left-adjust);
expand	make the table as wide as the current line length;
box	enclose the table in a box;
doublebox	enclose the table in a double box;
allbox	enclose each item of the table in a box;
tab (*x*)	use the character *x* instead of a tab to separate items in a line of input data.

The global options, if any, are terminated with a semi-colon (;).

Following the global options are lines describing the format of each line of the table. Each such format line describes one line of the actual table, except that the last format line (which must end with a period) describes all remaining lines of the table. Each column of each line of the table is described by a single keyletter, optionally followed by specifiers. Specifiers govern formatting aspects such as the font and point size of the corresponding item, where vertical bars are to appear between columns, column width, and inter-column spacing. The available keyletters are:

c	center item within the column;
r	right-adjust item within the column;
l	left-adjust item within the column;
n	numerically adjust item in the column: units positions of numbers are aligned vertically;
s	span previous item on the left into this column;
a	center longest line in this column and then left-adjust all other lines in this column with respect to that centered line;
^	span down previous entry in this column;
_	replace this entry with a horizontal line;
=	replace this entry with a double horizontal line.

The characters **B** and **I** stand for the bold and italic fonts, respectively; the character | indicates a vertical line between columns.

The format lines are followed by lines containing the actual data for the table. Within such data lines, data items are normally separated by tab characters. The end of the data items is indicated by a line containing only **.TE**.

If a data line consists of only _ or =, a single or double line, respectively, is drawn across the table at that point; if a *single item* in a data line consists of only _ or =, then that item is replaced by a single or double line.

Full details of all these and other features of *tbl* are given in the reference manual cited below.

The – TX option forces *tbl* to use only full vertical line motions, making the output more suitable for devices that cannot generate partial vertical line motions (e.g., line printers).

If no filenames are given as arguments (or if – is specified as the last argument), *tbl* reads the standard input, so it may be used as a filter. When it is used with *eqn*(1) or *neqn*, *tbl* should

come first to minimize the volume of data passed through pipes.

EXAMPLE

If → represents a tab (which should be typed as a genuine tab), then the input:

```
.TS
center box ;
cB s s
cI | cI s
^ | c c
l | n n .
Household Population
_
Town→Households
→Number→Size
=
Bedminster→789→3.26
Bernards Twp.→3087→3.74
Bernardsville→2018→3.30
Bound Brook→3425→3.04
Bridgewater→7897→3.81
Far Hills→240→3.19
.TE
```

yields:

Household Population		
Town	*Households*	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Bridgewater	7897	3.81
Far Hills	240	3.19

SEE ALSO

cw(1), eqn(1), mm(1), mmt(1), nroff(1), troff(1), mm(5), mv(5).
"Table Formatting Program" in the *Document Processing Guide*.

BUGS

See *BUGS* under *nroff*(1).

NAME

tc – phototypesetter simulator

SYNOPSIS

tc [**–t**] [**–sn**] [**–pl**] [file]

DESCRIPTION

Tc interprets its input (standard input default) as device codes for a Wang Laboratories, Inc. C/A/T phototypesetter. The standard output of *tc* is intended for a Tektronix 4014 terminal with ASCII and APL character sets. The sixteen typesetter sizes are mapped into the 4014's four sizes; the entire TROFF character set is drawn using the 4014's character generator, with overstruck combinations where necessary. Typical usage is:

troff –t files | tc

At the end of each page, *tc* waits for a new-line (empty line) from the keyboard before continuing to the next page. In this wait state, the command **e** suppresses the screen erase before the next page; **s***n* causes the next *n* pages to be skipped; and **!***cmd* sends *cmd* to the shell.

The command line options are:

– **t** Don't wait between pages (for directing output into a file).

– **s***n* Skip the first *n* pages.

– **p***l* Set page length to *l*; *l* may include the scale factors **p** (points), **i** (inches), **c** (centimeters), and **P** (picas); default is picas.

SEE ALSO

4014(1), sh(1), tplot(1G), troff(1).

BUGS

Font distinctions are lost.

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** option ignores interrupts; the **– a** option 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, returns a zero (true) exit status; otherwise, a non-zero (false) exit status is returned; *test* also returns a non-zero exit status if there are no arguments. 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.
– **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 that all the operators and flags are separate arguments to *test*. Notice also that parentheses are meaningful to the shell and, therefore, must be escaped.

SEE ALSO

find(1), sh(1).

WARNING

In the second form of the command (i.e., the one that uses [ ], rather than the word *test*), the square brackets must be delimited by blanks.

Some UNIX systems do not recognize the second form of the command.

NAME

time - time a command

SYNOPSIS

time command

DESCRIPTION

The *command* is executed; after it is complete, *time* prints the time elapsed during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on standard error.

SEE ALSO

timex(1), times(2).

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.

Options are:

- **–p** List process accounting records for *command* and all its children. Suboptions **f, h, k, m, r,** and **t** modify the data items reported, as defined in *acctcom*(1). 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.
- **–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.

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
```

SEE ALSO

acctcom(1), 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 are spuriously included.

NAME

tip, cu – connect to a remote system

SYNOPSIS

tip [**–v**] [*–speed*] system-name
tip [**–v**] [*–speed*] phone-number
cu phone-number [**–t**] [**–s** *speed*] [**-a** *acu*] [**–l** *line*] [**–#**]

DESCRIPTION

Tip and *cu* establish a full-duplex connection to another machine, giving the appearance of being logged in directly on the remote cpu. It goes without saying that you must have a login on the machine (or equivalent) to which you wish to connect. The preferred interface is *tip*. The *cu* interface is included for those people attached to the "call UNIX" command of version 7. This manual page describes only *tip*.

Typed characters are normally transmitted directly to the remote machine (which does the echoing as well). A tilde ('~') appearing as the first character of a line is an escape signal; the following are recognized:

~^D ~. Drop the connection and exit (you may still be logged in on the remote machine).

~c [*name*] Change directory to name (no argument implies change to your home directory).

~! Escape to a shell (exiting the shell will return you to tip).

~> Copy file from local to remote. *Tip* prompts for the name of a local file to transmit.

~< Copy file from remote to local. *Tip* prompts first for the name of the file to be sent, then for a command to be executed on the remote machine.

~p *from* [*to*]
Send a file to a remote UNIX host. The put command causes the remote UNIX system to run the command string "cat > 'to'", while *tip* sends it the "from" file. If the "to" file isn't specified the "from" file name is used. This command is actually a UNIX specific version of the "~>" command.

~t *from* [*to*]
Take a file from a remote UNIX host. As in the put command the "to" file defaults to the "from" file name if it isn't specified. The remote host executes the command string "cat 'from';echo ^A" to send the file to *tip*.

~| Pipe the output from a remote command to a local UNIX process. The command string sent to the local UNIX system is processed by the shell.

~# Send a BREAK to the remote system. For systems which don't support the necessary *ioctl* call the break is simulated by a sequence of line speed changes and DEL characters.

~s Set a variable (see the discussion below).

~^Z Stop *tip* (only available with job control).

~? Get a summary of the tilde escapes

Tip uses the file /etc/remote to find how to reach a particular system and to find out how it should operate while talking to the system; refer to *remote*(4) for a full description. Each system has a default baud rate with which to establish a connection. If this value is not suitable, the baud rate to be used may be specified on the command line, e.g. "tip -300 mds".

When *tip* establishes a connection it sends out a connection message to the remote system; the default value, if any, is defined in /etc/remote.

When *tip* prompts for an argument (e.g. during setup of a file transfer) the line typed may be edited with the standard erase and kill characters. A null line in response to a prompt, or an interrupt, will abort the dialogue and return you to the remote machine.

Tip guards against multiple users connecting to a remote system by opening modems and terminal lines with exclusive access, and by honoring the locking protocol used by *uucp*(1C).

During file transfers *tip* provides a running count of the number of lines transferred. When using the ~> and ~< commands, the "eofread" and "eofwrite" variables are used to recognize end-of-file when reading, and specify end-of-file when writing (see below). File transfers normally depend on tandem mode for flow control. If the remote system does not support tandem mode, "echocheck" may be set to indicate *tip* should synchronize with the remote system on the echo of each transmitted character.

When *tip* must dial a phone number to connect to a system it will print various messages indicating its actions. *Tip* supports the DEC DN-11 and Racal-Vadic 831 auto-call-units; the DEC DF02 and DF03, Ventel 212+, Racal-Vadic 3451, and Bizcomp 1031 and 1032 integral call unit/modems.

VARIABLES

Tip maintains a set of *variables* which control its operation. Some of these variable are read-only to normal users (root is allowed to change anything of interest). Variables may be displayed and set through the "s" escape. The syntax for variables is patterned after *vi*(1) and *Mail*(1). Supplying "all" as an argument to the set command displays all variables readable by the user. Alternatively, the user may request display of a particular variable by attaching a '?' to the end. For example "escape?" displays the current escape character.

Variables are numeric, string, character, or boolean values. Boolean variables are set merely by specifying their name; they may be reset by prepending a '!' to the name. Other variable types are set by concatenating an '=' and the value. The entire assignment must not have any blanks in it. A single set command may be used to interrogate as well as set a number of variables. Variables may be initialized at run time by placing set commands (without the "~s" prefix in a file *.tiprc* in one's home directory). The **-v** option causes *tip* to display the sets as they are made. Certain common variables have abbreviations. The following is a list of common variables, their abbreviations, and their default values.

beautify
(bool) Discard unprintable characters when a session is being scripted; abbreviated *be*.

baudrate
(num) The baud rate at which the connection was established; abbreviated *ba*.

dialtimeout
(num) When dialing a phone number, the time (in seconds) to wait for a connection to be established; abbreviated *dial*.

echocheck
(bool) Synchronize with the remote host during file transfer by waiting for the echo of the last character transmitted; default is *off*.

eofread
(str) The set of characters which signify and end-of-tranmission during a ~< file transfer command; abbreviated *eofr*.

eofwrite
(str) The string sent to indicate end-of-transmission during a ~> file transfer command; abbreviated *eofw*.

eol
(str) The set of characters which indicate an end-of-line. *Tip* will recognize escape

characters only after an end-of-line.

escape

(char) The command prefix (escape) character; abbreviated *es*; default value is '~'.

exceptions

(str) The set of characters which should not be discarded due to the beautification switch; abbreviated *ex*; default value is "\t\n\f\b".

force

(char) The character used to force literal data transmission; abbreviated *fo*; default value is '^P'.

framesize

(num) The amount of data (in bytes) to buffer between file system writes when receiving files; abbreviated *fr*.

host

(str) The name of the host to which you are connected; abbreviated *ho*.

prompt

(char) The character which indicates and end-of-line on the remote host; abbreviated *pr*; default value is '\n'. This value is used to synchronize during data transfers. The count of lines transferred during a file transfer command is based on receipt of this character.

raise

(bool) Upper case mapping mode; abbreviated *ra*; default value is *off*. When this mode is enabled, all lower case letters will be mapped to upper case by *tip* for transmission to the remote machine.

raisechar

(char) The input character used to toggle upper case mapping mode; abbreviated *rc*; default value is '^A'.

record

(str) The name of the file in which a session script is recorded; abbreviated *rec*; default value is "tip.record".

script

(bool) Session scripting mode; abbreviated *sc*; default is *off*. When *script* is *true*, *tip* will record everything transmitted by the remote machine in the script record file specified in *record*. If the *beautify* switch is on, only printable ASCII characters will be included in the script file (those characters betwee 040 and 0177). The variable *exceptions* is used to indicate characters which are an exception to the normal beautification rules.

tabexpand

(bool) Expand tabs to spaces during file transfers; abbreviated *tab*; default value is *false*. Each tab is expanded to 8 spaces.

verbose

(bool) Verbose mode; abbreviated *verb*; default is *true*. When verbose mode is enabled, *tip* prints messages while dialing, shows the current number of lines transferred during a file transfer operations, and more.

SHELL

(str) The name of the shell to use for the ~! command; default value is "/bin/sh", or taken from the environment.

HOME

(str) The home directory to use for the ~c command; default value is taken from the

environment.

FILES

/etc/remote	global system descriptions
/etc/phones	global phone number data base
${REMOTE}	private system descriptions
${PHONES}	private phone numbers
~/.tiprc	initialization file.
/usr/spool/uucp/LCK..*	lock file to avoid conflicts with *uucp*

DIAGNOSTICS

Diagnostics are, hopefully, self explanatory.

SEE ALSO

remote(4), phones(4)

BUGS

The full set of variables is undocumented and should, probably, be pared down.

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. 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).

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.

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).

BUGS

Won't handle ASCII NUL in *string1* or *string2*; always deletes NUL from input.

NAME

troff – typeset text

SYNOPSIS

troff [options] [files]

DESCRIPTION

Troff formats text contained in *files* (standard input by default). Its capabilities are described in the "NROFF and TROFF User's Manual" section of the *Document Processing Guide*.

An argument consisting of a minus (–) is taken to be a file name corresponding to the standard input. The *options*, which may appear in any order, but must appear before the *files*, are:

- **–o***list* Print only pages whose page numbers appear in the *list* of numbers and ranges, separated by commas. A range *N*–*M* means pages *N* through *M*; an initial –*N* means from the beginning to page *N*; and a final *N*– means from *N* to the end. (See *BUGS* below.)
- **–n***N* Number first generated page *N*.
- **–s***N* Stop every *N* pages. *Troff* will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and resume when the typesetter's start button is pressed.
- **–r***aN* Set register *a* (which must have a one-character name) to *N*.
- **–i** Read standard input after *files* are exhausted.
- **–q** Invoke the simultaneous input-output mode of the **.rd** request.
- **–z** Print only messages generated by **.tm** (terminal message) requests.
- **–m***name* Prepend to the input *files* the non-compacted (ASCII text) macro file **/usr/lib/tmac/tmac.***name*.
- **–c***name* Prepend to the input *files* the compacted macro files **/usr/lib/macros/cmp.**[**nt**].[**dt**].*name* and **/usr/lib/macros/ucmp.**[**nt**].*name*.
- **–k***name* Compact the macros used in this invocation of *troff*, placing the output in files [**dt**].*name* in the current directory (see the "NROFF/TROFF User's Manual" in the *Document Processing Guide* for details of compacting macro files).
- **–t** Direct output to the standard output instead of the phototypesetter.
- **–f** Refrain from feeding out paper and stopping phototypesetter at the end of the run.
- **–w** Wait until phototypesetter is available, if it is currently busy.
- **–b** Report whether the phototypesetter is busy or available. No text processing is done.
- **–a** Send a printable ASCII approximation of the results to the standard output.
- **–p***N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elapsed time.
- **–g** Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output (this option is not usable on most systems). This option is not compatible with the **–s** option; furthermore, when this option is invoked, all **.fp** (font position) requests (if any) in the *troff* input must come before the first break, and *no* **.tl** requests may come before the first break.
- **–T***name* Use font-width tables for device *name* (the font tables are found in **/usr/lib/font/***name***/***). Currently, no *names* are supported.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta$#	temporary file
/usr/lib/tmac/tmac.*	standard macro files and pointers
/usr/lib/macros/*	standard macro files
/usr/lib/font/*	font width tables for *troff*

SEE ALSO

"NROFF/TROFF User's Manual" in the *Document Processing Guide*.

cw(1), eqn(1), mmt(1), nroff(1), tbl(1), tc(1), mm(5), mv(5).

BUGS

Troff believes in Eastern Standard Time; as a result, depending on the time of the year and on your local time zone, the date that *troff* generates may be off by one day from your idea of what the date is.

When *troff* is used with the **–o***list* option inside a pipeline (e.g., with one or more of *cw*(1), *eqn*(1), and *tbl*(1)), it may cause a harmless "broken pipe" diagnostic if the last page of the document is not specified in *list*.

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) such as:

```
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

Tsort 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.

BUGS

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty – get the terminal's name

SYNOPSIS

tty [**–l**] [**–s**]

DESCRIPTION

Tty prints the pathname of the user's terminal. The **–l** option prints the synchronous line number to which the user's terminal is connected, if it is on an active synchronous line. The **–s** option inhibits printing of the terminal's pathname, allowing one to test just the exit code.

EXIT CODES

2	if invalid options were specified,
0	if standard input is a terminal,
1	otherwise.

DIAGNOSTICS

not on an active synchronous line means the standard input is not a synchronous terminal and the **–l** option is specified.

not a tty means the standard input is not a terminal and the **–s** option is not specified.

NAME

ucbstty – set terminal options

SYNOPSIS

ucbstty [option ...]

DESCRIPTION

Ucbstty sets certain I/O options on the current output terminal, placing its output on the diagnostic output. With no argument, it reports the speed of the terminal and the settings of the options which are different from their defaults. With the argument "all", all normally used option settings are reported. With the argument "everything", everything *ucbstty* knows about is printed. The option strings are selected from the following set:

even allow even parity input
– even disallow even parity input
odd allow odd parity input
– odd disallow odd parity input
raw raw mode input (**no** input processing (erase, kill, interrupt, ...); parity bit passed back)
– raw negate raw mode
cooked same as '– raw'
cbreak make each character available to *read*(2) as received; no erase and kill processing, but all other processing (interrupt, suspend, ...) is performed
– cbreak make characters available to *read* only when newline is received
– nl allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl accept only new-line to end lines
echo echo back every character typed
– echo do not echo characters
lcase map upper case to lower case
– lcase do not map case
tandem enable flow control, so that the system sends out the stop character when its internal queue is in danger of overflowing on input, and sends the start character when it is ready to accept further input
– tandem disable flow control
– tabs replace tabs by spaces when printing
tabs preserve tabs
ek set erase and kill characters to # and @

For the following commands which take a character argument *c*, you may also specify *c* as the "u" or "undef", to set the value to be undefined. A value of "^x", a 2 character sequence, is also interpreted as a control character, with "^?" representing delete.

erase *c* set erase character to *c* (default '#', but often reset to ^H.)
kill *c* set kill character to *c* (default '@ ', but often reset to ^U.)
intr *c* set interrupt character to *c* (default DEL or ^? (delete), but often reset to ^C.)
quit *c* set quit character to *c* (default control \.)
start *c* set start character to *c* (default control Q.)
stop *c* set stop character to *c* (default control S.)
eof *c* set end of file character to *c* (default control D.)
brk *c* set break character to *c* (default undefined.) This character is an extra wakeup causing character.
cr0 cr1 cr2 cr3
select style of delay for carriage return (see *ioctl*(2))
nl0 nl1 nl2 nl3
select style of delay for linefeed

tab0 tab1 tab2 tab3
select style of delay for tab

ff0 ff1 select style of delay for form feed

bs0 bs1 select style of delay for backspace

tty33 set all modes suitable for the Teletype Corporation Model 33 terminal.

tty37 set all modes suitable for the Teletype Corporation Model 37 terminal.

vt05 set all modes suitable for Digital Equipment Corp. VT05 terminal

dec set all modes suitable for Digital Equipment Corp. operating systems users; (erase, kill, and interrupt characters to ^?, ^U, and ^C, decctlq and "newcrt".)

tn300 set all modes suitable for a General Electric TermiNet 300

ti700 set all modes suitable for Texas Instruments 700 series terminal

tek set all modes suitable for Tektronix 4014 terminal

0 hang up phone line immediately

50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb
Set terminal baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

A teletype driver which supports the job control processing of *csh*(1) and more functionality than the basic driver is fully described in *tty*(4). The following options apply only to it.

new Use new driver (switching flushes typeahead).

crt Set options for a CRT (crtbs, ctlecho and, if >= 1200 baud, crterase and crtkill.)

crtbs Echo backspaces on erase characters.

prterase For printing terminal echo erased characters backwards within "\" and "/".

crterase Wipe out erased characters with "backspace-space-backspace."

– crterase Leave erased characters visible; just backspace.

crtkill Wipe out input on like kill ala **crterase.**

– crtkill Just echo line kill character and a newline on line kill.

ctlecho Echo control characters as "^*x*" (and delete as "^?".) Print two backspaces following the EOT character (control D).

– ctlecho Control characters echo as themselves; in cooked mode EOT (control-D) is not echoed.

decctlq After output is suspended (normally by ^S), only a start character (normally ^Q) will restart it. This is compatible with DEC's vendor supplied systems.

– decctlq After output is suspended, any character typed will restart it; the start character will restart output without providing any input. (This is the default.)

tostop Background jobs stop if they attempt terminal output.

– tostop Output from background jobs to the terminal is allowed.

tilde Convert "~" to "`" on output (for Hazeltine terminals).

– tilde Leave poor "~" alone.

flusho Output is being discarded usually because user hit control O (internal state bit).

– flusho Output is not being discarded.

pendin Input is pending after a switch from cbreak to cooked and will be re-input when a read becomes pending or more input arrives (internal state bit).

– pendin Input is not pending.

intrup Send a signal (SIGTINT) to the terminal control process group whenever an input record (line in cooked mode, character in cbreak or raw mode) is available for reading.

– intrup Don't send input available interrupts.

mdmbuf Start/stop output on carrier transitions (not implemented).

– mdmbuf
Return error if write attempted after carrier drops.

litout Send output characters without any processing.
– litout Do normal output processing, inserting delays, etc.
nohang Don't send hangup signal if carrier drops.
– nohang Send hangup signal to control process group when carrier drops.
etxack Diablo style etx/ack handshaking (not implemented).

The following special characters are applicable only to the new teletype driver and are not normally changed.

susp *c* set suspend process character to *c* (default control Z).
dsusp *c* set delayed suspend process character to *c* (default control Y).
rprnt *c* set reprint line character to *c* (default control R).
flush *c* set flush output character to *c* (default control O).
werase *c* set word erase character to *c* (default control W).
lnext *c* set literal next character to *c* (default control V).

SEE ALSO

stty(1), ioctl(2), tabs(1), tset(1), ucbtty(7), termio(7)

NAME

ul – do underlining

SYNOPSIS

ul [**– i**] [**– t** *terminal*] [*name ...*]

DESCRIPTION

Ul reads the named files (or standard input if none are given) and translates occurrences of underscores to the sequence which indicates underlining for the terminal in use, as specified by the environment variable TERM. The **– t** option overrides the terminal kind specified in the environment. The file */etc/termcap* is read to determine the appropriate sequences for underlining. If the terminal is incapable of underlining, but is capable of a standout mode then that is used instead. If the terminal can overstrike, or handles underlining automatically, *ul* degenerates to *cat*(1). If the terminal cannot underline, underlining is ignored.

The **– i** option causes *ul* to indicate underlining onto by a separate line containing appropriate dashes '– '; this is useful when you want to look at the underlining which is present in an *nroff* output stream on a crt-terminal.

SEE ALSO

man(1), nroff(1), colcrt(1)

BUGS

Nroff usually outputs a series of backspaces and underlines intermixed with the text to indicate underlining. No attempt is made to optimize the backward motion.

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.

SEE ALSO

chmod(1), sh(1), chmod(2), creat(2), umask(2).

NAME

uname – print name of current UNIX System

SYNOPSIS

uname [– **snrvma**]

DESCRIPTION

Uname prints the current system name of the UNIX System on the standard output file. It is mainly useful to determine what system one is using. The options cause selected information returned by *uname*(2) to be printed:

– **s** print the system name (default).

– **n** print the nodename (the nodename may be a name that the system is known by to a communications network).

– **r** print the operating system release.

– **v** print the operating system version.

– **m** print the machine hardware name.

– **a** print all the above information.

Arguments not recognized default the command to the – **s** option.

SEE ALSO

uname(2).

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 prior to 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).

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 and compares 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 some less familiar units, 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 thus: **brgallon**. For a complete list of units, type:

```
cat /usr/lib/unittab
```

FILES

/usr/lib/unittab

NAME

users – compact list of users who are on the system

SYNOPSIS

users

DESCRIPTION

Users lists the login names of the users currently on the system in a compact, one-line format.

FILES

/etc/utmp

SEE ALSO

who(1)

NAME

uucp, uulog, uuname – unix to unix copy

SYNOPSIS

uucp [options] source-files destination-file

uulog [options]

uuname [**–l**]

DESCRIPTION

Uucp.

Uucp copies files named by the *source-file* arguments to the *destination-file* argument. A filename 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 which *uucp* knows about. The *system-name* may also be a list of names such as

system-name!system-name!...!system-name!path-name

in which case an attempt is made to send the file via the specified route, and only to a destination in PUBDIR (see below). Care should be taken to insure that intermediate nodes in the route are willing to foward information.

The shell metacharacters **?**, ***** and **[...]** appearing in *path-name* are expanded on the appropriate system.

Pathnames may be one of:

(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 *~/user* where *user* is a login name on the specified system and is replaced by that user's directory under PUBDIR;

(4) anything else is prefixed by the current directory.

If the result is an erroneous pathname for the remote system the copy fails. 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*:

–d Make all necessary directories for the file copy (default).

–f Do not make intermediate directories for the file copy.

–c Use the source file when copying out rather than copying the file to the spool directory (default).

–C Copy the source file to the spool directory.

–m*file* Report status of the transfer in *file*. If *file* is omitted, send mail to the requester when the copy is completed.

–n*user*
Notify *user* on the remote system that a file was sent.

–e*sys* Send the *uucp* command to system *sys* to be executed there. (Note: this is only successful if the remote machine allows the *uucp* command to be executed by **/usr/lib/uucp/uuxqt**.)

Uucp returns on the standard output a string which is the job number of the request. This job number can be used by *uustat* to obtain status or terminate the job.

Uulog.

Uulog queries a summary log of *uucp* and *uux*(1C) transactions in the file **/usr/spool/uucp/LOGFILE.**

The options cause *uulog* to print logging information:

– **s***sys* Print information about work involving system *sys*.

– **u***user*

Print information about work done for the specified *user*.

Uuname.

Uuname lists the uucp names of known systems. The – **l** option returns the local system name.

FILES

/usr/spool/uucp	spool directory
/usr/spool/uucppublic	public directory for receiving and sending (PUBDIR)
/usr/lib/uucp/*	other data and program files

SEE ALSO

mail(1), uux(1C).

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You are probably not able to fetch files by pathname; ask a responsible person on the remote system to send them to you. For the same reasons you are probably not able to send files to arbitrary pathnames. As distributed, the remotely accessible files are those whose names begin **/usr/spool/uucppublic** (equivalent to ~**nuucp** or just ~).

BUGS

All files received by *uucp* are then owned by *uucp*.

The – **m** option only works for sending files or receiving a single file. Receiving multiple files specified by special shell characters ? * [...] does not activate the – **m** option.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

uustat [options]

DESCRIPTION

Uustat displays the status of, or cancels, previously specified *uucp* commands, or provides general status on *uucp* connections to other systems. The following *options* are recognized:

– **j***jobn* Report the status of the *uucp* request *jobn*. If **all** is used for *jobn*, the status of all *uucp* requests is reported. If *jobn* is omitted, the status of the current user's *uucp* requests is reported.

– **k***jobn* Kill the *uucp* request whose job number is *jobn*. The killed *uucp* request must belong to the person issuing the *uustat* command unless one is the superuser.

– **r***jobn* Rejuvenate *jobn*. *Jobn* is touched so that its modification time is set to the current time. This prevents *uuclean* from deleting the job until the jobs modification time reaches the limit imposed by *uuclean*.

– **c***hour* Remove status entries older than *hour* hours. This administrative option can only be initiated by the user **uucp** or the superuser.

– **u***user* Report the status of all *uucp* requests issued by *user*.

– **s***sys* Report the status of all *uucp* requests that communicate with remote system *sys*.

– **o***hour* Report the status of all *uucp* requests which are older than *hour* hours.

– **y***hour* Report the status of all *uucp* requests which are younger than *hour* hours.

– **m***mch* Report the status of accessibility of machine *mch*. If *mch* is specified as **all**, then the status of all machines known to the local *uucp* is provided.

– **M***mch* This is the same as the – *m* option except that two times are printed: the time that the last status was obtained and the time that the last successful transfer to that system occurred.

– **O** Report the *uucp* status using the octal status codes listed below. If this option is not specified, the verbose description is printed with each *uucp* request.

– **q** List the number of jobs and other control files queued for each machine and the time of the oldest and youngest file queued for each machine. If a lock file exists for that system, its date of creation is listed.

When no options are given, *uustat* outputs the status of all *uucp* requests issued by the current user. Note that only one of the options – **j**, – **m**, – **k**, – **c**, – **r**, can be used with the rest of the other options.

For example, the command:

uustat – uhdc – smhtsa – y72

prints the status of all *uucp* requests that were issued by user *hdc* to communicate with system *mhtsa* within the last 72 hours. The meaning of the job request status statement is:

job-number user remote-system command-time status-time status

where the *status* may be either an octal number or a verbose description. The octal code corresponds to the following description:

OCTAL	STATUS
000001	the copy failed, but the reason cannot be determined
000002	permission to access local file is denied
000004	permission to access remote file is denied
000010	bad *uucp* command is generated
000020	remote system cannot create temporary file
000040	cannot copy to remote directory

000100	cannot copy to local directory
000200	local system cannot create temporary file
000400	cannot execute *uucp*
001000	copy (partially) succeeded
002000	copy finished, job deleted
004000	job is queued
010000	job killed (incomplete)
020000	job killed (complete)

The meaning of the machine accessibility status statement is:

system-name time status

where *time* is the latest status time and *status* is a self-explanatory description of the machine status.

FILES

/usr/spool/uucp	spool directory
/usr/lib/uucp/L_stat	system status file
/usr/lib/uucp/R_stat	request status file

SEE ALSO

uucp(1C).

NAME

uuto, uupick – public UNIX System-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 the user's machine. Destination has the form:

system!*user*

where *system* is taken from a list of system names that *uucp* knows about (see *uuname*). *Logname* 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. 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:

<new-line>	Go on to next entry.
d	Delete the entry.
m [*dir*]	Move the entry to named directory *dir* (current directory is default).
a [*dir*]	Same as **m** except moving all the files sent from *system*.
p	Print the content of the file.
q	Stop.
EOT (control-d)	Same as **q**.
!*command*	Escape to the shell to do *command*.
*	Print a command summary.

Uupick invoked with the **–s***system* option only searches the PUBDIR for files sent from *system*.

FILES

PUBDIR	/usr/spool/uucppublic	public directory

SEE ALSO

mail(1), uuclean(1M), uucp(1C), uustat(1C), uux(1C).

NAME

uux – unix to unix command execution

SYNOPSIS

uux [options] command-string

DESCRIPTION

Uux gathers zero or more files from various systems, executes a command on a specified system and then sends standard output to a file on a specified system. Note that, for security reasons, many installations limit the list of commands executable on behalf of an incoming request from *uux*. Many sites permit little more than the receipt of mail (see *mail*(1)) via *uux*.

The *command-string* is made up of one or more arguments that look like a shell command line, except that the command names and filenames may be prefixed by *system-name*!. A null *system-name* is interpreted as the local system.

Filenames 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) prefixed by the current directory.

As an example, the command

uux "!diff usg!/usr/dan/f1 pwba!/a4/dan/f1 > !f1.diff"

gets the **f1** files from the "usg" and "pwba" machines, executes a *diff* command and puts the results in **f1.diff** in the local 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 attempts to get all files to the execution system. For output files, the filename must be escaped using parentheses. For example, the command

uux a!uucp b!/usr/file \(c!/usr/file\)

sends a *uucp* command to system "a" to get **/usr/file** from system "b" and send it to system "c".

Uux notiies you if the requested command on the remote system is disallowed. 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*.

– **n** Send no notification to user.

– **m***file* Report status of the transfer in *file*. If *file* is omitted, send mail to the requester when the copy is completed.

Uux returns an ASCII string on the standard output which is the job number. This job number can be used by *uustat* to obtain the status or terminate a job.

FILES

/usr/spool/uucp	spool directory
/usr/lib/uucp/*	other data and programs

SEE ALSO

uuclean(1M), uucp(1C).

BUGS

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.

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 end-of-file 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 keyletter arguments are defined as follows. 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* (*S*CCS *ID*entification String) 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, etc., which may exist) or invalid (e.g., **r1**.0 or **r1**.1.0 is 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.

– **m***name* The argument value *name* is compared with the SCCS %M% keyword in *file*.

– **y***type* 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 follows:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;
bit 2 = corrupted SCCS file;
bit 3 = can't 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).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

Val can process up to 50 files on a single command line. Any number above 50 produces 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.
– c*char*	Specifies a control character to be used in place of **:**.
– **s**	Silences warning (not error) messages 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 without 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

> 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, and 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
Print the given message on the diagnostic output.

:err message
Print 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.

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*] [**+***command*] [**– l**] [**– w***n*] [**– x**] name ...

DESCRIPTION

Vi (visual) 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.

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. The "Vi Quick Reference" card and the "Introduction to Display Editing with Vi" in the *User's Guide* provide full details on using *vi*.

INVOCATION

The following invocation options are interpreted by *vi*:

– t*tag*	Edit the file containing the *tag* and position the editor at its definition.
– r*file*	Recover *file* after an editor or system crash. If *file* is not specified a list of all saved files is printed.
+*command*	Begin editing by executing the specified editor search or positioning *command* .
– l	**LISP** mode; indents appropriately for lisp code, the () {} [[and]] commands in *vi* and *open* are modified to have meaning for *lisp* .
– w*n*	Set the default window size to *n*. This is useful when using the editor over a slow speed line.
– x	Encryption mode; a key is prompted for allowing creation or editing of an encrypted file.

The *name* argument indicates files to be edited.

"*VI STATES*"

Command	Normal and initial state. Other states return to command state upon completion. ESC (escape) is used to cancel a partial command.
Insert	Entered by **a i A I o O c C s S R.** Arbitrary text may then be entered. Insert is normally terminated with ESC character, or abnormally with interrupt.
Last line	Reading input for **:** **/** **?** or **!**; terminate with ESC or CR to execute, interrupt to cancel.

COMMANDS

Counts before vi commands

line/column number	**z G |**
scroll amount	**^D ^U**
replicate insert	**a i A I**
repeat effect	most of the rest

Sample commands

dw	delete a word
de	... leaving white space
dd	delete a line
3dd	... 3 lines
i*text*ESC	insert text *abc*
cw*new*ESC	change word to *new*
ea*s*ESC	pluralize word
xp	transpose characters

ZZ	exit vi

Interrupting, canceling

ESC	end insert or incomplete cmd
^?	(delete or rubout) interrupts
^L	reprint screen if **^?** scrambles it

File manipulation

:w	write back changes
:wq	write and quit
:q	quit
:q!	quit, discard changes
:e *name*	edit file *name*
:e!	reedit, discard changes
:e + *name*	edit, starting at end
:e +*n*	edit starting at line *n*
:e #	edit alternate file
^	synonym for **:e #**
:w *name*	write file *name*
:w! *name*	overwrite file *name*
:sh	run shell, then return
:!*cmd*	run *cmd*, then return
:n	edit next file in arglist
:n *args*	specify new arglist
:f	show current file and line
^G	synonym for **:f**
:ta *tag*	to tag file entry *tag*
^]	**:ta**, following word is *tag*

Positioning within file

^F	forward screen
^B	backward screen
^D	scroll down half screen
^U	scroll up half screen
G	goto line (end default)
/pat	next line matching *pat*
? *pat*	prev line matching *pat*
n	repeat last / or ?
N	reverse last / or ?
/pat/+n	n'th line after *pat*
? *pat***?−** *n*	n'th line before *pat*
]]	next section/function
[[	previous section/function
%	find matching ( ) { or }

Adjusting the screen

^L	clear and redraw
^R	retype, eliminate @ lines
zCR	redraw, current at window top
z−	... at bottom
z.	... at center
*/pat/z−**	*pat* line at bottom
z*n***.**	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and returning

``	previous context
''	... at first non-white in line
m*x*	mark position with letter *x*
`*x*	to mark *x*
'*x*	... at first non-white in line

Line positioning

H	home window line
L	last window line
M	middle window line
+	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
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^**H**	same as ←
space	same as →
f*x*	find *x* forward
F*x*	**f** backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last **f F t** or **T**
,	inverse of ;
|	to specified column
%	find matching ({) or }

Words, sentences, paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back **W**
E	to end of **W**

Commands for LISP Mode

)	Forward s-expression
}	... but don't stop at atoms
(	Back s-expression
{	... but don't stop at atoms

Corrections during insert

^**H**	erase last character
^**W**	erase last word
erase	your erase, same as ^**H**

kill	your kill, erase input this line
\	escapes ^H, your erase and kill
ESC	ends insertion, back to command
^?	interrupt, terminates insert
^D	backtab over *autoindent*
↑^D	kill *autoindent*, save for next
0^D	... but at margin next also
^V	quote non-printing character

Insert and replace

a	append after cursor
i	insert before
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R	replace characters

Operators (double to affect lines)

d	delete
c	change
<	left shift
>	right shift
!	filter through command
=	indent for LISP
y	yank lines to buffer

Miscellaneous operations

C	change rest of line
D	delete rest of line
s	substitute chars
S	substitute lines
J	join lines
x	delete characters
X	... before cursor
Y	yank lines

Yank and put

p	put back lines
P	put before
"*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

SEE ALSO

ex (1).

"Vi Quick Reference" card. "An Introduction to Display Editing with Vi" in the *Document Processing Guide*.

WARNINGS AND BUGS

Software tabs using ^T work only immediately after the *autoindent*.

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The *wrapmargin* option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Saving text on deletes in the named buffers is somewhat inefficient.

The *source* command does not work when executed as **:source**; there is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a **:** escape. To use these on a **:global** you must **Q** to *ex* command mode, execute them, and then reenter the screen editor with *vi* or *open*.

NAME

vmstat– report virtual memory statistics

SYNOPSIS

vmstat - qmpdsAv [interval [count]]

DESCRIPTION

Vmstat delves into the system and reports certain statistics kept about virtual memory performance. Unless otherwise stated, these statistics are updated every 5 seconds by the system. The statistics printed by the various flags are:

- **– q** number of procs on runq, waiting on disk i/o, waiting on paging, and sleeping in core
- **– m** total and active virtual and real memory, number of currently free pages, avg free pages over last 5 and 30 seconds
- **– p** page faults, page dups (from copy-on-write forks), pageins, intransit blocking pageins, total page reclaims, page reclaims from free list, page "reclaims" from buffer cache, pages paged in, pages filled on demand from executables, pages zero filled on demand
- **– d** pageout daemon activity including pages scanned, revolutions of hand, pageouts, pages paged out, pages freed
- **– s** swapins, pages swapped in, swapouts, pages swapped out
- **– A** all of the above – **v** visual mode using terminfo capabilities to refresh the display.

If *interval* is supplied, vmstat will sleep for *interval* seconds and read/print the structs again. If *count* is supplied, this will happen *count* times, otherwise this will continue until interrupted.

FILES

/dev/kmem, /unix

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait until all processes started with & have completed, and report on abnormal terminations.

Because the *wait*(2) system call must be executed in the parent process, the shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(1).

BUGS

This command cannot wait for processes of a 3-or-more-stage pipeline that are not children of the shell.

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 new-lines.

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 are printed along with the counts.

NAME

what - identify SCCS files

SYNOPSIS

what 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 ", >, new-line, \, 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
```

prints

```
f.c:
        identification information
f.o:
        identification information
a.out:
        identification information
```

What is intended to be used in conjunction with the command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually.

SEE ALSO

get(1), help(1).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

It's possible that an unintended occurrence of the pattern **@(#)** could be found just by chance, but this causes no harm in nearly all cases.

NAME

whatis – describe what a command is

SYNOPSIS

whatis command ...

DESCRIPTION

Whatis looks up a given command and gives the header line from the manual section. You can then run the *man*(1) command to get more information. If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. Try 'whatis ed' and then you should do 'man 1 ed' to get the manual.

Whatis is actually just the **–f** option to the *man*(1) command.

FILES

/usr/man/whatis Data base

SEE ALSO

man(1).

NAME

whereis – locate source, binary, and or manual for program

SYNOPSIS

whereis [– **sbm**] [– **u**] [– **SBM** dir ... – **f**] name ...

DESCRIPTION

Whereis locates source/binary and manuals sections for specified files. The supplied names are first stripped of leading pathname components and any (single) trailing extension of the form ".ext", e.g. ".c". Prefixes of "s." resulting from use of source code control are also dealt with. *Whereis* then attempts to locate the desired program in a list of standard places. If any of the – **b,** – **s** or – **m** flags are given then *whereis* searches only for binaries, sources or manual sections respectively (or any two thereof). The – **u** flag may be used to search for unusual entries. A file is said to be unusual if it does not have one entry of each requested type. Thus "whereis -m -u *" asks for those files in the current directory which have no documentation.

Finally, the – **B** – **M** and – **S** flags may be used to change or otherwise limit the places where *whereis* searches. The – **f** file flags is used to terminate the last such directory list and signal the start of file names.

EXAMPLE

The following finds all the files in /usr/bin which are not documented in /usr/man/man1 with source in /usr/src/cmd:

```
cd /usr/ucb
whereis – u – M /usr/man/man1 – S /usr/src/cmd – f *
```

FILES

/usr/src/*
/usr/{doc,man}/*
/lib, /etc, /usr/{lib,bin,ucb,local}

BUGS

Since the program uses *chdir*(2) to run faster, pathnames given with the – **M** – **S** and – **B** must be full; i.e. they must begin with a "/".

NAME

which - identify the full path name for a program using ATH

SYNOPSIS

which [name] ...

DESCRIPTION

Which takes a list of names and looks for the files which would be executed had these names been given as commands. Each argument is searched for along the user's path.

NAME

who – who is on the system

SYNOPSIS

who [– **uTlpdbrtas**] [file]

who am i

DESCRIPTION

Who lists 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 to obtain its information. If *file* is given, that file is examined. Usually, *file* is **/etc/wtmp**, which contains a history of all the logins since the file was last created.

Who with the **am i** option identifies the invoking user.

Except for the default – **s** option, the general format for output entries is:

name [state] line time activity pid [comment] [exit]

With options, *who* lists logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the *init* process. These options are:

– **u** List information about 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 *activity* is 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 twenty-four 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 whether a person is working at the terminal or not. 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 file can contain information about where the terminal is located, the telephone number of the dataset, type of terminal if hard-wired, etc.

– **T** Print the *state* of the terminal line. 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** List 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 doesn't exist.

– **p** List 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 *activity* 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** Display all processes that have expired and have 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** Indicate the time and date of the last reboot.

– **r** Indicate the current *run-level* of the *init* process. Following the run-level and date information are three fields which indicate the current state, the number of times that state was previously entered, and the previous state.

-t Indicate the last change to the system clock (via the *date*(1) command) by **root**. See *su*(1).

-a Process **/etc/utmp** or the named *file* with all options turned on.

-s List only the *name*, *line* and *time* fields; this is the default.

FILES

/etc/utmp
/etc/wtmp
/etc/inittab

SEE ALSO

init(1M) in the *Administrator's Manual*.
date(1), login(1), mesg(1), su(1), wait(2), inittab(4), utmp(4).

NAME

whoami – print effective current user id

SYNOPSIS

whoami

DESCRIPTION

Whoami prints who you are. It works even if you are su'd, while 'who am i' does not since it uses /etc/utmp.

FILES

/etc/passwd Name data base

SEE ALSO

who (1)

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 your intended recipient. When it has successfully completed the connection, it sends two bells to your 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 or an interrupt is sent. 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 is to be connected (e.g., **tty00**); otherwise, the first instance of the user found in **/etc/utmp** is assumed and the following message is 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, in particular *nroff*(1) and *pr*(1) disallow messages in order 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), nroff(1), pr(1), sh(1), who(1).

DIAGNOSTICS

user not logged in means the person you are trying to *write* to is not logged in.

NAME

wsplit - create RSD windows

SYNOPSIS

wsplit [-l] [w1] ... [w5] [L1] ... [L5]

DESCRIPTION

Wsplit is used to split the RSD into one or more different sized windows.

-l Causes the window(s) to be produced with no labels.

w[1-5] Specifies the widths of the new windows. Each value must be less than 1 and w1 + w2 + w3 + w4 + w5 <= 1. The default is .5 and .5 thus dividing the screen into 2 equal halves.

L[1-5] Gives each window a label (unless the -l option is given). Note that if less than 5 windows are desired, then window labels cannot be decimal values less than 1; otherwise, they will be mistaken for window width parameters. The default window label is their process id.

NAME

wtty – set window modes

SYNOPSIS

wtty [[-]mode] ... [spec [val]] ... [all]

DESCRIPTION

Wtty sets or displays window modes, size, and position parameters. The following *modes* can be enables. If preceded by a minus sign, the the *mode* is disabled.

scroll Causes the text in the window to scroll up when at bottom of the window.

wrap Causes the line to wrap around when end of line is reached.

csr Enables the cursor.

label The label at the top of the window appears.

save Textual data in an obscured window is saved and then restored when the window is no longer obscured.

bflsp Sets big font line spacing.

Spec parameters allow setting window size and position. The *val* parameter specifies either a coordinate or dimension in pixels.

uclx X-coordinate of upper left-hand corner of the window.

ulcy Y-coordinate of upper left-hand corner of the window.

width Specifies the width of the window.

height Specifies the height of the window.

all Displays the current modes, postions and size of the window.

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.

Xargs uses one's $PATH to search for *command*, which may be a shell file. 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 new-lines; 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 quoted strings a backslash (\) escapes 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 is 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** and – **n** are both given), 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* is with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first new-line *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 are 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) executes 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 none of the options **– i**, **– l**, or **– n** is 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.

– e*eofstr* *Eofstr* is taken as the logical end-of-file string. Underbar (_) is assumed for the logical EOF string if **– e** is not coded. **– e** with no *eofstr* coded turns off the logical EOF string capability (underbar is taken literally). *Xargs* reads standard input until either end-of-file or the logical EOF string is encountered.

Xargs terminates if it receives a return code of **– 1** from *command* 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 command moves all files from directory $1 to directory $2, and echoes each move command just before doing it:

```
ls $1 | xargs – i – t mv $1/{} $2/{}
```

The following command combines 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. They are archived 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 command executes *diff*(1) with successive pairs of arguments originally typed as shell arguments:

```
echo $* | xargs – n2 diff
```

DIAGNOSTICS

Self-explanatory.

NAME

xstr - extract strings from C programs to implement shared strings

SYNOPSIS

xstr [**-c**] [-] [file]

DESCRIPTION

Xstr maintains a file *strings* into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

xstr -c name

will extract the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file. The resulting C text is placed in the file *x.c*, to then be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffices of existing strings do not cause changes to the data base.

After all components of a large program have been compiled a file *xs.c* declaring the common *xstr* space can be created by a command of the form

xstr

This *xs.c* file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. A command

xstr name

creates files *x.c* and *xs.c* as before, without using or affecting any *strings* file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *Xstr* reads from its standard input when the argument '- ' is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

```
cc -E name.c |xstr -c -
cc -c x.c
mv x.o name.o
```

Xstr does not touch the file *strings* unless new items are added, thus *make* can avoid remaking *xs.o* unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array 'xstr'
/tmp/xs*	Temp file when 'xstr name' doesn't touch *strings*

SEE ALSO

mkstr(1)

AUTHOR

William Joy

BUGS

If a string is a suffix of another string in the data base, but the shorter string is seen first by *xstr* both strings will be placed in the data base, when just placing the longer one there will do.

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**– vdlt**] grammar

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes a 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** does 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 is 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 is not included. Program size is smaller and execution time is slightly faster when a program is produced without the runtime debugging code.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp	temporary file
yacc.debug	temporary file
yacc.acts	temporary file
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).

"YACC - Yet Another Compiler Compiler" in the *System V Support Tools 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.

BUGS

Because filenames are fixed, at most one *yacc* process can be active in a given directory at a time.

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always – 1; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

All the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

2 ENOENT No such file or directory

This error occurs when a filename is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 ESRCH No such process

No process can be found corresponding to that specified by the process identifier (*pid*) in *kill*(2) or *ptrace*(2).

4 EINTR Interrupted system call

An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error

Some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address

I/O on a special file refers to a subdevice which does not exist; or the I/O is beyond the limits of the device. This error may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than 5,120 bytes is presented to a member of the *exec*(2) family.

8 ENOEXEC Exec format error

A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(4)).

9 EBADF Bad file number

Either a file descriptor refers to no open file or a read (respectively write) request is made to a file which is open only for writing (respectively reading).

10 ECHILD No child processes

A *wait*(2) was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
A *fork*(2) failed because the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough space
During an *exec*(2), *brk*(2), or *sbrk*(2) call, a program asked for more space than the system is able to supply. This is not a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*(2).

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required
A non-block file was mentioned where a block device was required, e.g., in *mount*(2).

16 EBUSY Mount device busy
An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). This error also occurs if an attempt is made to enable accounting when it is already enabled.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link*(2).

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required; e.g., in a path prefix or as an argument to *chdir*(2).

21 EISDIR Is a directory
An attempt was made to write on a directory.

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*(2), or *kill*(2); reading or writing a file for which *lseek*(2) has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

23 ENFILE File table overflow
The system's table of open files is full, and temporarily *open*(2) cannot be accepted.

24 EMFILE Too many open files
No process may have more than 20 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
An attempt was made to execute a pure-procedure program which is currently open for writing or reading. This error also indicates an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT; see *ulimit*(2).

28 ENOSPC No space left on device
During a *write*(2) to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek*(2) was issued to a pipe.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt was made to make more than the maximum number of links (1000) to a file.

32 EPIPE Broken pipe
An attempt was made to write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is not representable within machine precision.

35 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).

36 EIDRM Identifier Removed
This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see *msgctl*(2), *semctl*(2), and *shmctl*(2)).

DEFINITIONS

Process ID
Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID
A new process is created by a currently active process; see *fork*(2). The parent process ID of a process is the process ID of its creator.

Process Group ID
Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(2).

Tty Group ID
Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group; see *exit*(2) and *signal*(2).

Real User ID and Real Group ID
Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group-ID bit set; see *exec*(2).

Superuser

A process is recognized as a *superuser* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

Filename.

Names consisting of 1 to 14 characters may be used to name an ordinary file, special file, or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of filenames because of the special meaning attached to these characters by the shell; see *sh*(1). Although permitted, it is advisable to avoid the use of unprintable characters in filenames.

Pathname and Path Prefix

A pathname is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a filename.

More precisely, a pathname is a null-terminated character string constructed as follows:

<pathname>::=<filename>|<path-prefix><filename>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/

where <*filename*> is a string of 1 to 14 characters other than the ASCII slash and null, and <*dirname*> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a pathname begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null pathame is treated as if it named a non-existent file.

Directory.

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot*, respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory.

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving pathname searches. A process's root directory need not be the root directory of the root file system.

File Access Permissions.

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The process's effective user ID is superuser.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

If none of these conditions exists, the corresponding permissions are denied.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;  /* operation permission struct */
ushort   msg_qnum;           /* number of msgs on q */
ushort   msg_qbytes;         /* max number of bytes on q */
ushort   msg_lspid;          /* pid of last msgsnd operation */
ushort   msg_lrpid;          /* pid of last msgrcv operation */
time_t   msg_stime;          /* last msgsnd time */
time_t   msg_rtime;          /* last msgrcv time */
time_t   msg_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Msg_perm is an **ipc_perm** structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort  cuid;    /* creator user id */
ushort  cgid;    /* creator group id */
ushort  uid;     /* user id */
ushort  gid;     /* group id */
ushort  mode;    /* r/w permission */
```

Msg_qnum is the number of messages currently on the queue. *Msg_qbytes* is the maximum number of bytes allowed on the queue. *Msg_lspid* is the process id of the last process that performed a *msgsnd* operation (see *msgop*(2)). *Msg_lrpid* is the process id of the last process that performed a *msgrcv* operation (see *msgop*(2)). *Msg_stime* is the time of the last *msgsnd* operation, *msg_rtime* is the time of the last *msgrcv* operation, and *msg_ctime* is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions.

In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as {*token*}, where *token* is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a msqid are granted to a process if one or more of the following are true:

The process's effective user ID is superuser.

The process's effective user ID matches *msg_perm.[c]uid* in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of *msg_perm.mode* is set.

The process's effective user ID does not match *msg_perm.[c]uid*, the process's effective group ID matches *msg_perm.[c]gid*, and the appropriate bit of the "group" portion (060) of *msg_perm.mode* is set.

The process's effective user ID does not match *msg_perm.[c]uid*, the process's effective group ID does not match *msg_perm.[c]gid*, and the appropriate bit of the "other" portion (06) of *msg_perm.mode* is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;  /* operation permission struct */
ushort   sem_nsems;          /* number of sems in set */
time_t   sem_otime;          /* last operation time */
time_t   sem_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Sem_perm is an **ipc_perm** structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   cuid;       /* creator user id */
ushort   cgid;       /* creator group id */
ushort   uid;        /* user id */
ushort   gid;        /* group id */
ushort   mode;       /* r/a permission */
```

The value of *sem_nsems* is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. *Sem_num* values run sequentially from 0 to the value of *sem_nsems* minus 1. *Sem_otime* is the time of the last *semop*(2) operation, and *sem_ctime* is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort   semval;     /* semaphore value */
short    sempid;     /* pid of last operation  */
ushort   semncnt;    /* # awaiting semval > cval */
ushort   semzcnt;    /* # awaiting semval = 0 */
```

Semval is a non-negative integer. *Sempid* is equal to the process ID of the last process that performed a semaphore operation on this semaphore. *Semncnt* is a count of the number of processes that are currently suspended until this semaphore's *semval* becomes greater than its current value. *Semzcnt* is a count of the number of processes that are currently suspended until this semaphore's *semval* becomes zero.

Semaphore Operation Permissions.

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation

is given as {*token*}, where *token* is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, Alter by group
00006	Read, Alter by others

Read and Alter permissions on a semid are granted to a process if one or more of the following are true:

The process's effective user ID is superuser.

The process's effective user ID matches *sem_perm.[c]uid* in the data structure associated with semid and the appropriate bit of the "user" portion (0600) of *sem_perm.mode* is set.

The process's effective user ID does not match *sem_perm.[c]uid*, the process's effective group ID matches *sem_perm.[c]gid*, and the appropriate bit of the "group" portion (060) of *sem_perm.mode* is set.

The process's effective user ID does not match *sem_perm.[c]uid*, the process's effective group ID does not match *sem_perm.[c]gid*, and the appropriate bit of the "other" portion (06) of *sem_perm.mode* is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct  ipc_perm shm_perm;  /* operation permission struct */
int     shm_segsz;          /* size of segment */
ushort  shm_cpid;           /* creator pid */
ushort  shm_lpid;           /* pid of last operation */
short   shm_nattch;         /* number of current attaches */
time_t  shm_atime;          /* last attach time */
time_t  shm_dtime;          /* last detach time */
time_t  shm_ctime;          /* last change time */
                            /* Times measured in secs since */
                            /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shm_perm is an **ipc_perm** structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort  cuid;       /* creator user id */
ushort  cgid;       /* creator group id */
ushort  uid;        /* user id */
ushort  gid;        /* group id */
ushort  mode;       /* r/w permission */
```

Shm_segsz specifies the size of the shared memory segment. *Shm_cpid* is the process id of the process that created the shared memory identifier. *Shm_lpid* is the process id of the last process that performed a *shmop*(2) operation. *Shm_nattch* is the number of processes that currently have this segment attached. *Shm_atime* is the time of the last *shmat* operation and *shm_dtime* is the time of the last *shmdt* operation; see *shmop*(2). *Shm_ctime* is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions.

In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an

operation is given as {*token*}, where *token* is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a shmid are granted to a process if one or more of the following are true:

The process's effective user ID is superuser.

The process's effective user ID matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of *shm_perm.mode* is set.

The process's effective user ID does not match *shm_perm.[c]uid*, the process's effective group ID matches *shm_perm.[c]gid*, and the appropriate bit of the "group" portion (060) of *shm_perm.mode* is set.

The process's effective user ID does not match *shm_perm.[c]uid*, the process's effective group ID does not match *shm_perm.[c]gid*, and the appropriate bit of the "other" portion (06) of *shm_perm.mode* is set.

Otherwise, the corresponding permissions are denied.

SEE ALSO

intro(3).

NAME

access - determine accessibility of a file

SYNOPSIS

int access (path, amode)
char *path;
int amode;

DESCRIPTION

Path points to a pathname naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

Read, write, or execute (search) permission is requested for a null pathname. [ENOENT]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits; members of the file's group other than the owner have permissions checked with respect to the "group" mode bits; all others have permissions checked with respect to the "other" mode bits.

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of - 1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), stat(2).

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record is written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(2). The effective user ID of the calling process must be superuser to use this call.

Path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

Acct fails if one or more of the following are true:

The effective user ID of the calling process is not superuser. [EPERM]

An attempt is made to enable accounting when it is already enabled. [EBUSY]

A component of the path prefix is not a directory. [ENOTDIR]

One or more components of the accounting file's pathname do not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The file named by *path* is not an ordinary file. [EACCES]

Mode permission is denied for the named accounting file. [EACCES]

The named file is a directory. [EISDIR]

The named file resides on a read-only file system. [EROFS]

Path points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

acct(4).

NAME

alarm – set a process's alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

Alarm instructs the calling process's alarm clock to send the signal SIGALRM to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2).

Alarm requests are not stacked; successive calls reset the calling process's alarm clock.

If *sec* is 0, any previously made alarm request is canceled.

RETURN VALUE

Alarm returns the amount of time previously remaining in the calling process's alarm clock.

SEE ALSO

pause(2), signal(2).

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

Brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

Brk and *sbrk* fail without making any change in the allocated space if one or more of the following are true:

The requested change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)). [ENOMEM]

The requested change would result in the break value being greater than or equal to the start address of any attached shared memory segment (see *shmop*(2)).

RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2).

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Path points to the pathname of a directory. *Chdir* causes the named directory to become the current working directory. The starting point for *path* searches for pathnames that do not begin with /.

Chdir fails and the current working directory remains unchanged if one or more of the following are true:

A component of the pathname is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the pathname. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

chroot(2).

NAME

chmod – change mode of file

SYNOPSIS

int chmod (path, mode)
char *path;
int mode;

DESCRIPTION

Path points to a pathname naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
02000	Set group ID on execution.
01000	Save text image after execution.
00400	Read by owner.
00200	Write by owner.
00100	Execute (or search if a directory) by owner.
00070	Read, write, execute (search) by group.
00007	Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be superuser to change the mode of a file.

If the effective user ID of the process is not superuser, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not superuser or the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing, mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time.

Chmod fails and the file mode remains unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not superuser. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

chown(2), mknod(2).

NAME

chown – change owner and group of a file

SYNOPSIS

int chown (path, owner, group)
char *path;
int owner, group;

DESCRIPTION

Path points to a pathname naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with the effective user ID equal to the file owner or superuser may change the ownership of a file.

If *chown* is invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode are cleared (bits 04000 and 02000, respectively). See *chmod*(2) for a complete list of access permission bits.

Chown fails and the owner and group of the named file remain unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not superuser. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2).

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a pathname naming a directory. *Chroot* causes the named directory to become the root directory. The starting point for *path* searches for pathnames that begin with /.

The effective user ID of the process must be superuser to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

Chroot fails and the root directory remains unchanged if one or more of the following are true:

Any component of the pathname is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not superuser. [EPERM]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

chdir(2).

NAME

close - close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call. *Close* closes the file descriptor indicated by *fildes*.

Close fails if *fildes* is not a valid open file descriptor. [EBADF]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of - 1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

int creat (path, mode)
char *path;
int mode;

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the pathname pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode*, modified as follows:

All bits set in the process's file mode creation mask are cleared; see *umask*(2).

Mode bit 01000 (save text image after execution) is cleared; see *chmod*(2).

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls; see *fcntl*(2). No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat fails if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The pathname is null. [ENOENT]

The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]

The named file resides or would reside on a read-only file system. [EROFS]

The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]

The file exists and write permission is denied. [EACCES]

The named file is an existing directory. [EISDIR]

Twenty (20) file descriptors are currently open. [EMFILE]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., the file descriptor) is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), dup(2), lseek(2), open(2), read(2), umask(2), write(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write, or read/write).

The new file descriptor is set to remain open across *exec*(2) system calls; see *fcntl*(2).

The file descriptor returned is the lowest one available.

Dup fails if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Twenty (20) file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion a non-negative integer (i.e., the file descriptor) is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

int execl (path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp)
char *path, *argv[ ], *envp [ ];

int execlp (file, arg0, arg1, ..., argn, 0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header (see *a.out*(4)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

main (*argc, argv, envp*)
int *argc*;
char ***argv, ***envp*;

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Path points to a pathname that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the calling process's environment in the global cell **extern char **environ;**. This pointer is used to pass the calling process's environment to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the

file pointer is unchanged.

Signals set to terminate the calling process are set to terminate the new process. Signals set to be ignored by the calling process are set to be ignored by the new process. Signals set to be caught by the calling process are set to terminate the new process; see *signal*(2).

If the set-user-ID mode bit of the new process file is set (see *chmod*(2)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process are not attached to the new process (see *shmop*(2)).

Profiling is disabled for the new process; see *profil*(2).

The new process also inherits the following attributes from the calling process:

nice value (see *nice*(2))
process ID
parent process ID
process group ID
semadj values (see *semop*(2))
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))

Exec fails and returns to the calling process if one or more of the following are true:

One or more components of the new process file's pathname do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

The new process requires more memory than is allowed by the system-imposed maximum MAXMEM. [ENOMEM]

The number of bytes in the new process's argument list is greater than the system-imposed limit of 5,120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

Path, *argv*, or *envp* points to an illegal address. [EFAULT]

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value is – 1 and *errno* is set to indicate the error.

SEE ALSO

exit(2), fork(2), environ(5).

NAME

exit, _exit – terminate process

SYNOPSIS

void exit (status)
int status;
void _exit (status)
int status;

DESCRIPTION

Exit terminates the calling process with the following consequences:

All the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low-order 8 bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table; it has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see <**sys/proc.h**>) to be used by *times*.

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(2)) inherits each of these processes.

Each attached shared memory segment is detached and the value of *shm_nattach* in the data structure associated with its shared memory identifier is decremented by 1; see *shmop*(2).

For each semaphore for which the calling process has set a semaphore adjustment (semadj) value (see *semop*(2)), that semadj value is added to the *semval* of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed (see *plock*(2)).

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(2).

If the process ID, tty group ID, and process group ID of the calling process are equal, the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

acct(2), plock(2), semop(2), shmop(2), signal(2), times(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

fchmod – change mode of a file descriptor

SYNOPSIS

int fchmod (fildes, mode)
int fildes;
int mode;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, or *fcntl* system call. *Fchmod* sets the access permission portion of the mode of the file associated with *fildes* according to the bit pattern contained in *mode*.

Fchmod operates identically to *chmod*; only the form of the first argument differs. Refer to *chmod* for a complete description.

SEE ALSO

chmod(2).

NAME

fchown – change owner and group of a file descriptor

SYNOPSIS

int fchown (fildes, owner, group)
int fildes;
int owner, group;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat, open, dup, fcntl,* or *pipe* system call. The owner ID and group ID of the file associated with *fildes* are set to the numeric values contained in *owner* and *group* respectively.

Fchown operates identically to *chown*; only the form of the first argument differs. Refer to *chown* for a complete description.

SEE ALSO

chown(2).

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides control over open files. *Fildes* is an open file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

The *cmd*s available are:

F_DUPFD — Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write, or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD — Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0**, the file remains open across *exec*; otherwise the file is closed upon execution of *exec*.

F_SETFD — Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL — Get *file* status flags.

F_SETFL — Set *file* status flags to *arg*. Only certain flags can be set; see *fcntl*(5).

Fcntl fails if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Cmd is F_DUPFD and 20 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and *arg* is negative or greater than 20. [EINVAL]

Refer to *fcntl*(5) for a list of the flag values contained in **<fcntl.h>**.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than – 1.
F_GETFL	Value of file flags.
F_SETFL	Value other than – 1.

Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), exec(2), open(2), fcntl(5).

NAME

fork – create a new process

SYNOPSIS

int fork ( )

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

- environment
- close-on-exec flag (see *exec*(2))
- signal handling settings (i.e., SIG_DFL, SIG_IGN, function address)
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status
- nice value (see *nice*(2))
- all attached shared memory segments (see *shmop*(2))
- process group ID
- tty group ID (see *exit*(2) and *signal*(2))
- trace flag (see *ptrace*(2) request 0)
- time left until an alarm clock signal (see *alarm*(2))
- current working directory
- root directory
- file mode creation mask (see *umask*(2))
- file size limit (see *ulimit*(2))

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared (see *semop*(2)).

Process locks, text locks, and data locks are not inherited by the child (see *plock*(2)).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0.

Fork fails and no child process is created if one or more of the following are true:

The system-imposed limit on the total number of processes under execution would be exceeded. [EAGAIN]

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. [EAGAIN]

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of – 1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exec(2), times(2), wait(2).

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

int getpid ( )

int getpgrp ( )

int getppid ( )

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

int getuid ( )

int geteuid ( )

int getgid ( )

int getegid ( )

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl – control device

SYNOPSIS

ioctl (fildes, request, arg)

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The descriptions of various devices in Section 7 of the *Administrator's Manual* discuss how *ioctl* applies to them.

Ioctl fails if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Fildes is not associated with a character special device. [ENOTTY]

Request or *arg* is not valid. See Section 7. [EINVAL]

RETURN VALUE

If an error has occurred, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

termio(7) in the *Administrator's Manual*.

NAME

kill - send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to the process or group of processes specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2) or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process unless the effective user ID of the sending process is superuser.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(2)) and are referenced below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* is sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* is sent to all processes, excluding *proc0* and *proc1*, whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not superuser, *sig* is sent to all processes, excluding *proc0* and *proc1*, whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is superuser, *sig* is sent to all processes, excluding *proc0* and *proc1*.

If *pid* is negative but not -1, *sig* is sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill fails and no signal is sent if one or more of the following are true:

Sig is not a valid signal number. [EINVAL]

No process can be found corresponding to that specified by *pid*. [ESRCH]

The user ID of the sending process is not superuser, and its real or effective user ID does not match the real or effective user ID of the receiving process. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2).

NAME

link – link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Path1 points to a pathname naming an existing file. *Path2* points to a pathname naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link fails and no link is created if one or more of the following are true:

A component of either path prefix is not a directory. [ENOTDIR]

A component of either path prefix does not exist. [ENOENT]

A component of either path prefix denies search permission. [EACCES]

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not superuser. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

Path2 points to a null pathname. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

unlink(2).

NAME

lseek - move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Fildes is a file descriptor returned from a *creat*(2), *open*(2), *dup*(2), or *fcntl*(2) system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek fails and the file pointer remains unchanged if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

Whence is not 0, 1, or 2. [EINVAL and SIGSYS signal]

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of - 1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

mknod – make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file named by the pathname pointed to by *path*. The mode of the new file is initialized from *mode*, where the value of *mode* is interpreted as follows:

0170000 file type; one of the following:
- 0010000 fifo special
- 0020000 character special
- 0040000 directory
- 0060000 block special
- 0100000 or 0000000 ordinary file

0004000 set user ID on execution
0002000 set group ID on execution
0001000 save text image after execution
0000777 access permissions; constructed from the following:
- 0000400 read by owner
- 0000200 write by owner
- 0000100 execute (search on directory) by owner
- 0000070 read, write, execute (search) by group
- 0000007 read, write, execute (search) by others

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask; all bits set in the process's file mode creation mask are cleared (see *umask*(2)). If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Mknod may be invoked only by the superuser for file types other than FIFO special.

Mknod fails and the new file is not created if one or more of the following are true:

The process's effective user ID is not superuser. [EPERM]

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(1), chmod(2), exec(2), umask(2), fs(4).

NAME

mount – mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to pathnames.

Upon successful completion, references to the file *dir* refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system. If the low-order bit is **1**, writing is forbidden; otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the superuser.

Mount fails if one or more of the following are true:

The effective user ID is not superuser. [EPERM]

Any of the named files does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

Dir is not a directory. [ENOTDIR]

Spec or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory, or is otherwise busy. [EBUSY]

The device associated with *spec* is currently mounted. [EBUSY]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

umount(2).

NAME

msgctl – message control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

DESCRIPTION

Msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *msg_perm.uid* in the data structure associated with *msqid*. Only superuser can raise the value of *msg_qbytes*.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *msg_perm.uid* in the data structure associated with *msqid*.

Msgctl fails if one or more of the following are true:

Msqid is not a valid message queue identifier. [EINVAL]

Cmd is not a valid command. [EINVAL]

Cmd is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser and is not equal to the value of *msg_perm.uid* in the data structure associated with *msqid*. [EPERM]

Cmd is equal to **IPC_SET,** an attempt is being made to increase to the value of *msg_qbytes,* and the effective user ID of the calling process is not equal to that of superuser. [EPERM]

Buf points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgget(2), msgop(2).

NAME

msgget – get message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

DESCRIPTION

Msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure (see *intro*(2)) are created for *key* if one of the following is true:

Key is equal to IPC_PRIVATE.

Key does not already have a message queue identifier associated with it, and (*msgflg* & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

Msg_perm.cuid, *msg_perm.uid*, *msg_perm.cgid*, and *msg_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *msg_perm.mode* are set equal to the low-order 9 bits of *msgflg*.

Msg_qnum, *msg_lspid*, *msg_lrpid*, *msg_stime*, and *msg_rtime* are set equal to 0.

Msg_ctime is set equal to the current time.

Msg_qbytes is set equal to the system limit.

Msgget fails if one or more of the following are true:

A message queue identifier exists for *key* but operation permission (see *intro*(2)), as specified by the low-order 9 bits of *msgflg*, would not be granted. [EACCES]

A message queue identifier does not exist for *key* and (*msgflg* & IPC_CREAT) is "false". [ENOENT]

A message queue identifier is to be created but the system imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded. [ENOSPC]

A message queue identifier exists for *key* but ((*msgflg* & IPC_CREAT) & (*msgflg* & IPC_EXCL)) is "true". [EEXIST]

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., a message queue identifier) is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgctl(2), msgop(2).

NAME

msgsnd, msgrcv – message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

Msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *Msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *Mtext* is any text of length *msgsz* bytes. *Msgsz* can range from 0 to a system imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to *msg_qbytes* (see *intro*(2)).

The total number of messages on all queues system-wide is equal to the system imposed limit.

These actions are as follows:

If (*msgflg* & IPC_NOWAIT) is "true", the message is not sent and the calling process returns immediately.

If (*msgflg* & IPC_NOWAIT) is "false", the calling process suspends execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM and a value of – 1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgsnd fails and no message is sent if one or more of the following are true:

Msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Mtype is less than 1. [EINVAL]

The message cannot be sent for one of the reasons cited above and (*msgflg* & IPC_NOWAIT) is "true". [EAGAIN]

Msgsz is less than zero or greater than the system imposed limit. [EINVAL]

Msgp points to an illegal address. [EFAULT]

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

Msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is the received message's type, as specified by the sending process. *Mtext* is the text of the message. *Msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & IPC_NOWAIT) is "true", the calling process returns immediately with a return value of – 1 and *errno* set to ENOMSG.

If (*msgflg* & IPC_NOWAIT) is "false", the calling process suspends execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of – 1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgrcv fails and no message is received if one or more of the following are true:

Msqid is not a valid message queue identifier. [EINVAL]

Operation permission is denied to the calling process. [EACCES]

Msgsz is less than 0. [EINVAL]

Mtext is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is "false". [E2BIG]

The queue does not contain a message of the desired type and (*msgtyp* & IPC_NOWAIT) is "true". [ENOMSG]

Msgp points to an illegal address. [EFAULT]

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

RETURN VALUES

If *msgsnd* or *msgrcv* returns due to the receipt of a signal, a value of − 1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of − 1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of − 1 is returned and *errno* is set to indicate the error.

SEE ALSO

msgctl(2), msgget(2).

NAME

nice – change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

Nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Nice fails and does not change the nice value if *incr* is negative and the effective user ID of the calling process is not superuser. [EPERM]

RETURN VALUE

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

nice(1), exec(2).

NAME

open – open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag, [mode] )
char *path;
int oflag, mode;
```

DESCRIPTION

Path points to a pathname naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only returns without delay. An *open* for writing-only returns an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only blocks until a process opens the file for writing. An *open* for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The *open* returns without waiting for carrier.

If O_NDELAY is clear:

The *open* blocks until carrier is present.

O_APPEND If set, the file pointer is set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

Mode bit 01000 (save text image after execution) is cleared. See *chmod*(2).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* fails if the file exists.

Upon successful completion a non-negative integer, the file descriptor, is returned.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

No process may have more than 20 file descriptors open simultaneously.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

20 file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the process's allocated address space. [EFAULT]

O_CREAT and O_EXCL are set and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., a file descriptor) is returned; otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

Refer to *fcntl*(5) for a list of the flag values contained in **<fcntl.h>**.

SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2), fcntl(5).

NAME

pause – suspend process until signal

SYNOPSIS

pause ( )

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* does not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function (see *signal*(2)), the calling process resumes execution from the point of suspension. A value of – 1 is returned from *pause* and *errno* is set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[**2**];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Writes up to 5,120 bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

Pipe fails if 19 or more file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

sh(1), read(2), write(2).

NAME

plock – lock process, text, or data in memory

SYNOPSIS

```
#include <sys/lock.h>

int plock (op)
int op;
```

DESCRIPTION

Plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *Plock* also allows these segments to be unlocked. The effective user ID of the calling process must be superuser to use this call. *Op* specifies the following:

PROCLOCK	lock text & data segments into memory (process lock)
TXTLOCK	lock text segment into memory (text lock)
DATLOCK	lock data segment into memory (data lock)
UNLOCK	remove locks

Plock fails and does not perform the requested operation if one or more of the following are true:

The effective user ID of the calling process is not superuser. [EPERM]

Op is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process. [EINVAL]

Op is equal to **TXTLOCK** and a text lock or a process lock already exists on the calling process. [EINVAL]

Op is equal to **DATLOCK** and a data lock or a process lock already exists on the calling process. [EINVAL]

Op is equal to **UNLOCK** and no type of lock exists on the calling process. [EINVAL]

RETURN VALUE

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2).

NAME

profil – execution time profile

SYNOPSIS

void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it and the result is multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling is turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), monitor(3C).

NAME

ptrace – process trace

SYNOPSIS

int ptrace (request, pid, addr, data);
int request, pid, addr, data;

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *sdb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for a list of signals), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. The parent also can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by the *func* argument of *signal*(2). The *pid*, *addr*, and *data* arguments are ignored and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated, request **1** returns a word from I space, and request **2** returns a word from D space. If I and D space are not separated, either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests fail if *addr* is not the start address of a word, in which case a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses in this area range from 0 to 2048. The *data* argument is ignored. This request fails if *addr* is not the start address of a word or is outside the USER area, in which case a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests fail if *addr* is a location in a pure procedure space and another process is executing in that space, or if *addr* is not the start address of a word. Upon failure a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are:

registers 0– 15

bits 0– 4 and 15 of the Processor Status Word

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals, including the one that caused the child to stop, are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal; any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request fails if *data* is not 0 or a valid signal number, in which case a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request sets the trace bit in the Processor Status Word of the child (bit 15) and then executes the same steps as listed above for request **7**. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.
Note: the trace bit is turned off after an interrupt.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it stops before executing the first instruction of the new image showing signal SIGTRAP.

GENERAL ERRORS

Ptrace in general fails if one or more of the following are true:

Request is an illegal number. [EIO]

Pid identifies a child that does not exist or has not executed a *ptrace* with request **0**. [ESRCH]

SEE ALSO

sdb(1), exec(2), signal(2), wait(2).

NAME

read – read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read returns a 0.

If O_NDELAY is clear, the read blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read returns a 0.

If O_NDELAY is clear, the read blocks until data becomes available.

Read fails if one or more of the following are true:

Fildes is not a valid file descriptor open for reading. [EBADF]

Buf points outside the allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), open(2), pipe(2), termio(7).

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort array[ ];
} arg;
```

DESCRIPTION

Semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum* (see *intro*(2) for definitions of values and permissions):

GETVAL Return the value of *semval*. {READ}

SETVAL Set the value of *semval* to *arg.val*. {ALTER} When this *cmd* is successfully executed, the semadj value (see *exit*(2)) corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of *sempid*. {READ}

GETNCNT Return the value of *semncnt*. {READ}

GETZCNT Return the value of *semzcnt*. {READ}

The following *cmd*s return and set, respectively, every *semval* in the set of semaphores.

GETALL Place *semval*s into array pointed to by *arg.array*. {READ}

SETALL Set *semval*s according to the array pointed to by *arg.array*. {ALTER} When this *cmd* is successfully executed, the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *sem_perm.uid* in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal

to either that of superuser or to the value of *sem_perm.uid* in the data structure associated with *semid*.

Semctl fails if one or more of the following are true:

Semid is not a valid semaphore identifier. [EINVAL]

Semnum is less than zero or greater than *sem_nsems*. [EINVAL]

Cmd is not a valid command. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is **SETVAL** or **SETALL** and the value to which *semval* is to be set is greater than the system imposed maximum. [ERANGE]

Cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser and is not equal to the value of *sem_perm.uid* in the data structure associated with *semid*. [EPERM]

Arg.buf points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of *semval*.
GETPID	The value of *sempid*.
GETNCNT	The value of *semncnt*.
GETZCNT	The value of *semzcnt*.
All others	A value of 0.

When *semctl* is unsuccessful, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

semget(2), semop(2), intro(2), exit(2).

NAME

semget – get set of semaphores

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

Semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores (see *intro*(2)) are created for *key* if one of the following is true:

Key is equal to IPC_PRIVATE.

Key does not already have a semaphore identifier associated with it, and (*semflg* & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *sem_perm.mode* are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and *sem_ctime* is set equal to the current time.

Semget fails if one or more of the following are true:

Nsems is either less than or equal to zero or greater than the system imposed limit. [EINVAL]

A semaphore identifier exists for *key* but operation permission (see *intro*(2)), as specified by the low-order 9 bits of *semflg*, would not be granted. [EACCES]

A semaphore identifier exists for *key* but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero. [EINVAL]

A semaphore identifier does not exist for *key* and (*semflg* & IPC_CREAT) is "false". [ENOENT]

A semaphore identifier is to be created but the system imposed limit on the maximum number of allowed semaphores system wide would be exceeded. [ENOSPC]

A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) & (*semflg* & IPC_EXCL)) is "true". [EEXIST]

RETURN VALUE

Upon successful completion, a non-negative integer (i.e., a semaphore identifier) is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

semctl(2), semop(2).

NAME

semop – semaphore operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf (*sops)[];
int nsops;
```

DESCRIPTION

Semop is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *Sops* is a pointer to the array of semaphore-operation structures. *Nsops* is the number of such structures in the array. Each structure includes the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

Sem_op specifies one of three semaphore operations as follows (see semaphore data structure in *intro*(2)):

If *sem_op* is a negative integer, one of the following occurs: {ALTER}

If *semval* is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* & SEM_UNDO) is "true", the absolute value of *sem_op* is added to the calling process's semadj value (see *exit*(2)) for the specified semaphore.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is "true", *semop* returns immediately.

If *semval* is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is "false", *semop* increments the *semncnt* associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

Semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* & SEM_UNDO) is "true", the absolute value of *sem_op* is added to the calling process's semadj value for the specified semaphore.

The *semid* for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, *errno* is set equal to EIDRM and a value of – 1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to *semval* and, if (*sem_flg* & SEM_UNDO) is "true", the value of *sem_op* is subtracted from the calling process's

semadj value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following occurs: {READ}

If *semval* is zero, *semop* returns immediately.

If *semval* is not equal to zero and (*sem_flg* & IPC_NOWAIT) is "true", *semop* returns immediately.

If *semval* is not equal to zero and (*sem_flg* & IPC_NOWAIT) is "false", *semop* increments the *semzcnt* associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of *semzcnt* associated with the specified semaphore is decremented.

The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM and a value of – 1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented and the calling process resumes execution in the manner prescribed in *signal*(2).

Semop fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

Semid is not a valid semaphore identifier. [EINVAL]

Sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*. [EFBIG]

Nsops is greater than the system imposed maximum. [E2BIG]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

The operation would result in suspension of the calling process but (*sem_flg* & IPC_NOWAIT) is "true". [EAGAIN]

The limit on the number of individual processes requesting a SEM_UNDO would be exceeded. [ENOSPC]

The number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the limit. [EINVAL]

An operation would cause a *semval* to overflow the system imposed limit. [ERANGE]

An operation would cause a semadj value to overflow the system imposed limit. [ERANGE]

Sops points to an illegal address. [EFAULT]

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of – 1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of – 1 is returned and *errno* is set to EIDRM.

Upon successful completion, the value of *semval* at the time of the call for the last operation in the array pointed to by *sops* is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), exec(2), exit(2), fork(2), semctl(2), semget(2).

NAME

setpgrp – set process group ID

SYNOPSIS

int setpgrp ( )

DESCRIPTION

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2).

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

Setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is superuser, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not superuser, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid (*setgid*) fails if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not superuser. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

getuid(2), intro(2).

NAME

shmctl – shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

DESCRIPTION

Shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT — Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET — Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *shm_perm.uid* in the data structure associated with *shmid*.

IPC_RMID — Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of *shm_perm.uid* in the data structure associated with *shmid*.

Shmctl fails if one or more of the following are true:

Shmid is not a valid shared memory identifier. [EINVAL]

Cmd is not a valid command. [EINVAL]

Cmd is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)). [EACCES]

Cmd is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser and is not equal to the value of *shm_perm.uid* in the data structure associated with *shmid*. [EPERM]

Buf points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

shmget(2), shmop(2).

NAME

shmget – get shared memory segment

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

DESCRIPTION

Shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of *size* bytes (see *intro*(2)) are created for *key* if one of the following is true:

Key is equal to IPC_PRIVATE.

Key does not already have a shared memory identifier associated with it, and (*shmflg* & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

Shm_perm.cuid, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of *shm_perm.mode* are set equal to the low-order 9 bits of *shmflg*. *Shm_segsz* is set equal to the value of *size*.

Shm_lpid, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.

Shm_ctime is set equal to the current time.

Shmget fails if one or more of the following are true:

Size is less than the system imposed minimum or greater than the system imposed maximum. [EINVAL]

A shared memory identifier exists for *key* but operation permission (see *intro*(2)), as specified by the low-order 9 bits of *shmflg*, would not be granted. [EACCES]

A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero. [EINVAL]

A shared memory identifier does not exist for *key* and (*shmflg* & IPC_CREAT) is "false". [ENOENT]

A shared memory identifier is to be created but the system imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded. [ENOSPC]

A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request. [ENOMEM]

A shared memory identifier exists for *key* but ((*shmflg* & IPC_CREAT) & (*shmflg* & IPC_EXCL)) is "true". [EEXIST]

RETURN VALUE

Upon successful completion a non-negative integer, i.e., a shared memory identifier, is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

shmctl(2), shmop(2).

NAME

shmat, shmdt – shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr
```

DESCRIPTION

Shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "false", the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if (*shmflg* & SHM_RDONLY) is "true" {READ}; otherwise it is attached for reading and writing {READ/WRITE}.

Shmat fails and does not attach the shared memory segment if one or more of the following are true:

Shmid is not a valid shared memory identifier. [EINVAL]

Operation permission is denied to the calling process (see *intro*(2)). [EACCES]

The available data space is not large enough to accommodate the shared memory segment. [ENOMEM]

Shmaddr is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address. [EINVAL]

Shmaddr is not equal to zero, (*shmflg* & SHM_RND) is "false", and the value of *shmaddr* is an illegal address. [EINVAL]

The number of shared memory segments attached to the calling process would exceed the system imposed limit. [EMFILE]

Shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

Shmdt fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment. [EINVAL]

RETURN VALUES

Upon successful completion, the return value is as follows:

Shmat returns the data segment start address of the attached shared memory segment.

Shmdt returns a value of 0.

Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), shmctl(2), shmget(2).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <sys/signal.h>
int (*signal (sig, func))()
int sig;
int (*func)();
```

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except SIGKILL:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	death of a child (see WARNING below)
SIGPWR	19	power fail (see WARNING below)

See below for the significance of the asterisk (*) in the above list.

Func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2); a "core image" is made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list and the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signal SIGKILL cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* is passed as the first argument to the signal-catching function. A second argument, *sig_code*, is also passed to the function. *Sig_code* has various contents, according to the value of *sig*. These values are provided in the table below. Before entering the signal-catching function, the value of *func* for the caught signal is set to SIG_DFL unless the signal is SIGILL, SIGTRAP, or SIGPWR.

Upon return from the signal-catching function, the receiving process resumes execution at the point it was interrupted. See the WARNINGS section below.

When a signal that is to be caught occurs during a *read*(2), *write*(2), *open*(2), or *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function is executed; then the interrupted system call returns a – 1 to the calling process with *errno* set to EINTR.

Note: the signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

Signal fails if one or more of the following are true:

Sig is an illegal signal number, including SIGKILL. [EINVAL]

Func points to an illegal address. [EFAULT]

The table below shows how handles M68010 traps. Most traps result in signals being sent to the user process that caused the trap. All other traps are considered to be STRAYFT, spurious interrupts.

The following meanings apply to information in the "SIGNAL CODE" column of the table:

code == address means the address causing the fault

code == pc means the program counter value at the time of the trap

code == (%d0) means the user parameter to the TRAP instruction

The definitions of KINTDIV, KINTOVF, and KSUBRNG are provided in the include file **<sys/signal.h>**.

TRAP TYPE	TRAP NO.	ASSIGNMENT	SIGNAL	SIGNAL CODE
BUSERR	2	bus error	SIGSEGV	address
		(see WARNINGS)or	SIGBUS	address
ADDRERR	3	address error	SIGILL	address
INSTERR	4	illegal instruction	SIGILL	pc
ZDVDERR	5	zero divide fault	SIGFPE	KINTDIV
CHKTRAP	6	CHK instruction fault	SIGFPE	KSUBRNG
TRAPVFT	7	TRAPV instruction fault	SIGFPE	KINTOVF
PRIVFLT	8	privileged instruction fault	SIGILL	pc
TRCTRAP	9	trace trap	SIGTRAP	pc
L1010FT	10	line 1010 emulator	SIGILLpc	
L1111FT	11	line 1111 emulator	SIGILL	pc
STRAYFT	24	spurious interrupt	n/a	n/a
SYSCALL	32	TRAP 0 - system call	n/a	(%d0)
BPTFLT	33	TRAP 1 - breakpoint	SIGTRAP	pc
IOTTRAP	34	TRAP 2 - simulate DEC IOT instruction	SIGIOT	(%d0)
EMTTRAP	35	TRAP 3 - simulate DEC EMT instruction	SIGEMT	(%d0)
FPETRAP	36	TRAP 4 - floating point exception	SIGFPE	(%d0)

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C).

WARNINGS

Two other signals that behave differently than the signals described above exist in this release of the system. They are:

SIGCLD	18	death of a child (reset when caught)
SIGPWR	19	power fail (not reset when caught)

There is no guarantee that, in future releases of the UNIX System, these signals will continue to behave as described below; they are included only for compatibility with other versions of the UNIX System. Their use in new programs is strongly discouraged.

For these signals, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL - ignore signal

The signal is to be ignored.

SIG_IGN - ignore signal

The signal is to be ignored. If *sig* is **SIGCLD**, the calling process's child processes do not create zombie processes when they terminate; see *exit*(2).

function address - catch signal

If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD**, except that, while the process is executing the signal-catching function, any received **SIGCLD** signals are queued and the signal-catching function is continually reentered

until the queue is empty.

The SIGCLD affects two other system calls (*wait*(2) and *exit*(2)) in the following ways:

wait If the *func* value of SIGCLD is set to SIG_IGN and a *wait* is executed, the *wait* blocks until all of the calling process's child processes terminate; it then returns a value of − 1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of SIGCLD is set to SIG_IGN, the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

Bus errors on the M68010 are mapped to either SIGBUS or SIGSEGV. If the offending address in invalid in the process address space (i.e. a wild pointer reference), then SIGSEGV is used. However, if the address is valid but merely causes a bus timeout (i.e. referencing a broken or non-existent controller), then SIGBUS is used.

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Path points to a pathname naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*. obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort   st_mode;  /* File mode; see mknod(2) */
ino_t    st_ino;   /* Inode number */
dev_t    st_dev;   /* ID of device containing */
                   /* a directory entry for this file */
dev_t    st_rdev;  /* ID of device */
                   /* This entry is defined only for */
                   /* character special or block */
                   /* special files */
short    st_nlink; /* Number of links */
ushort   st_uid;   /* User ID of the file's owner */
ushort   st_gid;   /* Group ID of the file's group */
off_t    st_size;  /* File size in bytes */
time_t   st_atime; /* Time of last access */
time_t   st_mtime; /* Time of last data modification */
time_t   st_ctime; /* Time of last file status change */
                   /* Times measured in seconds since */
                   /* 00:00:00 GMT, Jan. 1, 1970 */
```

St_atime, *st_mtime*, and *st_ctime* are changed by system calls as stated below.

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Stat fails if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Buf or *path* points to an invalid address. [EFAULT]

Fstat fails if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Buf points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), time(2), unlink(2).

NAME

stime – set time

SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Stime fails if the effective user ID of the calling process is not superuser. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(2).

NAME

sync – update super-block

SYNOPSIS

void sync ( )

DESCRIPTION

Sync causes all information in memory that should be on disk to be written out. This includes modified super-blocks, modified inodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*(1M) and *df*(1M). It is mandatory before a boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

SEE ALSO

Administrator's Manual.

NAME

time – get time

SYNOPSIS

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

Time fails if *tloc* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

stime(2).

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>

long times (buffer)
struct tms *buffer;
```

DESCRIPTION

Times fills the structure pointed to by *buffer* with time-accounting information. Contents of the structure are:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are in 60ths of a second.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

Tms_cstime is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

Times fails if *buffer* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of superuser may increase the limit. *Ulimit* fails and the limit remains unchanged if a process with an effective user ID other than superuser attempts to increase its file size limit. [EPERM]

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

brk(2), write(2).

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1), chmod(2), creat(2), mknod(2), open(2).

NAME

umount – unmount a file system

SYNOPSIS

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a pathname. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the superuser.

Umount fails if one or more of the following are true:

The process's effective user ID is not superuser. [EPERM]

Spec does not exist. [ENXIO]

Spec is not a block special device. [ENOTBLK]

Spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

Spec points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(2).

NAME

uname – get name of current operating system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current system in the structure pointed to by *name*.

Uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

Uname returns a null-terminated character string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the system is running on.

Uname fails if *name* points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

uname(1).

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the pathname pointed to by *path*.

The named file is unlinked unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Write permission is denied on the directory containing the link to be removed. [EACCES]

The named file is a directory and the effective user ID of the process is not superuser. [EPERM]

The entry to be unlinked is the mount point for a mounted file system. [EBUSY]

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1), close(2), link(2), open(2).

NAME

ustat - get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes the following elements:

```
daddr_t f_tfree;        /* Total free blocks */
ino_t   f_tinode;       /* Number of free inodes */
char    f_fname[6];     /* Filsys name */
char    f_fpack[6];     /* Filsys pack name */
```

Ustat fails if one or more of the following are true:

Dev is not the device number of a device containing a mounted file system. [EINVAL]

Buf points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of - 1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2), fs(4).

NAME

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

Path points to a pathname naming a file. *Utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the superuser may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct  utimbuf{
        time_t  actime;     /* access time */
        time_t  modtime;    /* modification time */
};
```

Utime fails if one or more of the following are true:

The named file does not exist. [ENOENT]

A component of the path prefix is not a directory. [ENOTDIR]

Search permission is denied by a component of the path prefix. [EACCES]

The effective user ID is not superuser and not the owner of the file and *times* is not NULL. [EPERM]

The effective user ID is not superuser and not the owner of the file, *times* is NULL, and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

Times is not NULL and points outside the process's allocated address space. [EFAULT]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2).

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

DESCRIPTION

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(2)), or until any one of the calling process's child processes stops in a trace mode (see *ptrace*(2)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called *status* are stored in the low-order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes. If the child process terminated, *status* identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high-order 8 bits of *status* contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177.

If the child process terminated due to an *exit* call, the low-order 8 bits of *status* are zero and the high-order 8 bits contain the low-order 8 bits of the argument that the child process passed to *exit*; see *exit*(2).

If the child process terminated due to a signal, the high-order 8 bits of *status* are zero and the low-order 8 bits contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal*(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see *intro*(2).

Wait fails and returns immediately if one or more of the following are true:

The calling process has no existing unwaited-for child processes. [ECHILD]

Stat_loc points to an illegal address. [EFAULT]

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of – 1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), pause(2), signal(2).

WARNING

See *WARNING* in *signal*(2).

NAME

write – write on a file

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND file status flag is set, the file pointer is set to the end of the file prior to each write.

Write fails and the file pointer remains unchanged if one or more of the following are true:

Fildes is not a valid file descriptor open for writing. [EBADF]

An attempt is made to write to a pipe that is not open for reading by any process. [EPIPE and SIGPIPE signal]

An attempt is made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2). [EFBIG]

Buf points outside the process's allocated address space. [EFAULT]

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for are written. For example, if there is space for 20 bytes more in a file before reaching a limit, a write of 512 bytes returns 20. The next write of a non-zero number of bytes gives a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes are permitted. Thus, the write fails if a write of *nbyte* bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) returns a count of 0. If O_NDELAY is clear, writes to a full pipe (or FIFO) block until space becomes available.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, – 1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library, *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **– lc** option. Some functions require declarations that can be included in the program being compiled by adding the line

#include <*header filename***>**

The appropriate #include file is indicated in the SYNOPSIS part of a function description.

(3F) These functions constitute the FORTRAN intrinsic function library, *libF77*. These functions are automatically available to the FORTRAN programmer and require no special invocation of the compiler.

(3M) These functions constitute the Math Library, *libm*. They are automatically loaded as needed by the FORTRAN compiler *f77*(1). They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **– lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**.

(3S) These functions constitute the "standard I/O package"; an introduction to this package is provided in *stdio*(3S). The functions are in the library *libc*, already mentioned. Declarations should be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

For descriptions and examples of #include files, refer to the "Libraries" section of the Programming Guide.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A NULL pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. NULL is defined as **0** in **<stdio.h>**; the user can include his own definition if he is not using **<stdio.h>**.

Many groups of FORTRAN intrinsic functions have *generic* function names that do not require explicit or implicit type declaration. The type of the function is determined by the type of its argument(s). For example, the generic function *max* returns an integer value if given integer arguments (*max0*), a real value if given real arguments (*amax1*), or a double-precision value if given double-precision arguments (*dmax1*).

FILES

/lib/libc.a

/usr/lib/libF77.a
/lib/libm.a

SEE ALSO

ar(1), cc(1), f77(1), ld(1), nm(1), intro(2), stdio(3S). *Programming Guide*.

DIAGNOSTICS

Functions in the Math Library (3M) may return the conventional values **0** or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE. Because many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to 6 characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, **/** for 1, **0** through **9** for 2– 11, **A** through **Z** for 12– 37, and **a** through **z** for 38– 63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than 6 characters, *a64l* uses the first 6.

L64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – generate an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

Abort causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO

adb(1), exit(2), kill(2), signal(2).

DIAGNOSTICS

If SIGIOT is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message **abort – core dumped** is written by the shell.

NAME

abort – terminate Fortran program

SYNOPSIS

call abort ( )

DESCRIPTION

Abort terminates the program which calls it, closing all open files truncated to the current position of the file pointer.

DIAGNOSTICS

When invoked, *abort* prints **Fortran abort routine called** on the standard error output.

SEE ALSO

abort(3C).

NAME

abs – return integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

BUGS

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

NAME

abs, iabs, dabs, cabs, zabs – Fortran absolute value

SYNOPSIS

```
integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2

r2 = abs(r1)

i2 = iabs(i1)
i2 = abs(i1)

dp2 = dabs(dp1)
dp2 = abs(dp1)

cx2 = cabs(cx1)
cx2 = abs(cx1)

dx2 = zabs(dx1)
dx2 = abs(dx1)
```

DESCRIPTION

Abs is the family of absolute value functions. *Iabs* returns the integer absolute value of its integer argument. *Dabs* returns the double-precision absolute value of its double-precision argument. *Cabs* returns the complex absolute value of its complex argument. *Zabs* returns the double-complex absolute value of its double-complex argument. The generic form *abs* returns the type of its argument.

SEE ALSO

floor(3M).

NAME

acos, dacos - Fortran arccosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **acos**(r1)

dp2 = **dacos**(dp1)
dp2 = **acos**(dp1)

DESCRIPTION

Acos returns the real arccosine of its real argument. *Dacos* returns the double-precision arccosine of its double-precision argument. The generic form *acos* may be used with impunity because its argument determines the type of the returned value.

SEE ALSO

trig(3M).

NAME

aimag, dimag – Fortran imaginary part of complex argument

SYNOPSIS

```
real r
complex cxr
double precision dp
double complex cxd
r = aimag(cxr)
dp = dimag(cxd)
```

DESCRIPTION

Aimag returns the imaginary part of its single-precision complex argument. *Dimag* returns the double-precision imaginary part of its double-complex argument.

NAME

aint, dint – Fortran integer part intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **aint**(r1)

dp2 = **dint**(dp1)
dp2 = **aint**(dp1)

DESCRIPTION

Aint returns the truncated value of its real argument in a real. *Dint* returns the truncated value of its double-precision argument as a double-precision value. *Aint* may be used as a generic function name, returning either a real or double-precision value depending on the type of its argument.

NAME

asin, dasin - Fortran arcsine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **asin**(r1)

dp2 = **dasin**(dp1)
dp2 = **asin**(dp1)

DESCRIPTION

Asin returns the real arcsine of its real argument. *Dasin* returns the double-precision arcsine of its double-precision argument. The generic form *asin* may be used with impunity as it derives its type from that of its argument.

SEE ALSO

trig(3M).

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. If *expression* is false (zero) when *assert* is executed, *assert* prints

Assertion failed: *expression, file xyz, line nnn*

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* is the source line number of the *assert* statement.

Compiling with the preprocessor option – DNDEBUG (see *cpp*(1)), or with the preprocessor control statement **#define** NDEBUG ahead of the **#include <assert.h>** statement, stops assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

atan, datan – Fortran arctangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **atan**(r1)

dp2 = **datan**(dp1)
dp2 = **atan**(dp1)

DESCRIPTION

Atan returns the real arctangent of its real argument. *Datan* returns the double-precision arctangent of its double-precision argument. The generic form *atan* may be used with a double-precision argument returning a double-precision value.

SEE ALSO

trig(3M).

NAME

atan2, datan2 – Fortran arctangent intrinsic function

SYNOPSIS

real r1, r2, r3
double precision dp1, dp2, dp3

r3 = **atan2**(r1, r2)

dp3 = **datan2**(dp1, dp2)
dp3 = **atan2**(dp1, dp2)

DESCRIPTION

Atan2 returns the arctangent of *arg1/arg2* as a real value. *Datan2* returns the double-precision arctangent of its double-precision arguments. The generic form *atan2* may be used with impunity with double-precision arguments.

SEE ALSO

trig(3M).

NAME

atof – convert ASCII string to floating-point number

SYNOPSIS

double atof (nptr)
char *nptr;

DESCRIPTION

Atof converts a character string pointed to by *nptr* to a double-precision floating-point number. The first unrecognized character ends the conversion. *Atof* recognizes an optional string of white-space characters (blanks or tabs), then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer. If the string begins with an unrecognized character, *atof* returns the value zero.

DIAGNOSTICS

When the correct value would overflow, *atof* returns HUGE, and sets *errno* to ERANGE. Zero is returned on underflow.

SEE ALSO

scanf(3S).

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

J0 and *j1* return Bessel functions of x of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of x of the first kind of order n.

Y0 and *y1* return the Bessel functions of x of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of x of the second kind of order n. The value of x must be positive.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1*, and *yn* to return the value HUGE and to set *errno* to EDOM. They also cause a message indicating DOMAIN error to be printed on the standard error output; the process will continue.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

and, or, xor, not, lshift, rshift - Fortran bitwise boolean functions

SYNOPSIS

```
integer i, j, k
real a, b, c
double precision dp1, dp2, dp3

k = and(i, j)
c = or(a, b)
j = xor(i, a)
j = not(i)
k = lshift(i, j)
k = rshift(i, j)
```

DESCRIPTION

The generic intrinsic boolean functions *and*, *or*, and *xor* return the value of the binary operations on their arguments. *Not* is a unary operator returning the one's complement of its argument. *Lshift* and *rshift* return the value of the first argument shifted left or right, respectively, the number of times specified by the second (integer) argument.

The boolean functions are generic, i.e., defined for all data types as arguments and return values. Where required, the compiler generates appropriate type conversions.

NOTE

Although defined for all data types, use of boolean functions on non-integer data is not productive.

BUGS

The implementation of the shift functions may cause large shift values to deliver unexpected results.

NAME

bsearch – binary search

SYNOPSIS

char *bsearch ((**char** *) key, (**char** *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to the datum to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

lsearch(3C), hsearch(3C), qsort(3C), tsearch(3C).

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

DESCRIPTION

Clock returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2) or *system*(3S).

The resolution of the clock is 16.667 milliseconds on M68000 or DEC processors.

SEE ALSO

times(2), wait(2), system(3S).

BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2,147 seconds of CPU time (about 36 minutes).

NAME

conjg, dconjg – Fortran complex conjugate intrinsic function

SYNOPSIS

complex cx1, cx2
double complex dx1, dx2

cx2 = **conjg**(cx1)

dx2 = **dconjg**(dx1)

DESCRIPTION

Conjg returns the complex conjugate of its complex argument. *Dconjg* returns the double-complex conjugate of its double-complex argument.

NAME

toupper, tolower, _toupper, _tolower, toascii – translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*(3S): the integers from – 1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C), getc(3S).

NAME

cos, dcos, ccos – Fortran cosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = cos(r1)
dp2 = dcos(dp1)
dp2 = cos(dp1)
cx2 = ccos(cx1)
cx2 = cos(cx1)
```

DESCRIPTION

Cos returns the real cosine of its real argument. *Dcos* returns the double-precision cosine of its double-precision argument. *Ccos* returns the complex cosine of its complex argument. The generic form *cos* may be used with impunity because its returned type is determined by that of its argument.

SEE ALSO

trig(3M).

NAME

cosh, dcosh – Fortran hyperbolic cosine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **cosh**(r1)

dp2 = **dcosh**(dp1)
dp2 = **cosh**(dp1)

DESCRIPTION

Cosh returns the real hyperbolic cosine of its real argument. *Dcosh* returns the double-precision hyperbolic cosine of its double-precision argument. The generic form *cosh* may be used to return the hyperbolic cosine in the type of its argument.

SEE ALSO

sinh(3M).

NAME

crypt, setkey, encrypt – generate DES encryption

SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, edflag)
char *block;
int edflag;
```

DESCRIPTION

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended to frustrate use of hardware implementations of the DES for key search.

Key is a user's typed password. *Salt* is a 2-character string chosen from the set [**a-zA-Z0-9./**]; this string is used to perturb the DES algorithm in one of 4,096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first 2 characters are the salt itself.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. The 56-bit key is used with the above-mentioned algorithm to encrypt or decrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

login(1), passwd(1), getpass(3C), passwd(4).

BUGS

The return value points to static data that is overwritten by each call.

NAME

ctermid – generate filename for terminal

SYNOPSIS

```
#include <stdio.h>

char *ctermid(s)
char *s;
```

DESCRIPTION

Ctermid generates the pathname of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the pathname is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the **<stdio.h>** header file.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a string (**/dev/tty**) that refers to the terminal if used as a filename. For this reason, *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to string

SYNOPSIS

```
#include <time.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

Ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

Localtime and *gmtime* return pointers to *tm* structures, described below. *Localtime* corrects for the time zone and possible Daylight Savings Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the system uses.

Asctime converts a *tm* structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the *tm* structure, are in the *<time.h>* header file. The structure declaration is:

```
struct tm {
        int tm_sec; /* seconds (0 - 59) */
        int tm_min; /* minutes (0 - 59) */
        int tm_hour; /* hours (0 - 23) */
        int tm_mday; /* day of month (1 - 31) */
        int tm_mon; /* month of year (0 - 11) */
        int tm_year; /* year - 1900 */
        int tm_wday; /* day of week (Sunday = 0) */
        int tm_yday; /* day of year (0 - 365) */
        int tm_isdst;
};
```

Tm_isdst is non-zero if Daylight Savings Time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if, and only if, the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named TZ is present, *asctime* uses the contents of the variable to override the default time zone. The value of TZ must be a 3-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional 3-letter name for a daylight time zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting TZ are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

```
char *tzname[2] = { "EST", "EDT" };
```

are set from the environment variable TZ. The function *tzset* sets these external variables from TZ; *tzset* is called by *asctime* and may also be called explicitly by the user.

Note that in most installations, TZ is set by default when the user logs on, to a value in the local /etc/profile file (see *profile*(4)).

SEE ALSO

time(2), getenv(3C), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

. . .

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (– 1); see *stdio*(3S)).

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, similar to *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

ascii(5).

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

Cuserid generates a character-string representation of the login name of the owner of the current process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character **(\0)** is placed at *s[0]*.

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL *call;

void undial (fd)
int fd;
```

DESCRIPTION

Dial returns a file descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <**dial.h**> header file.

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The CALL typedef in the <**dial.h**> header file is:

```
typedef struct {
        struct termio   *attr;  /* pointer to termio attribute struct */
        int             baud;   /* transmission data rate */
        int             speed;  /* 212A modem: low=300, high=1200 */
        char            *line;  /* device name for out-going line */
        char            *telno;         /* pointer to tel-no digits string */
        int             modem;          /* specify modem control for direct lines */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high-speed or low-speed setting on the 212A modem. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200).

If the desired terminal line is a direct line, a string pointer to its device name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *L-devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *L-devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described on the *acu*(7). The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the <**termio.h**> header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is important for attributes such as parity and baud rate.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2).
termio(7) in the *Administrator's Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure is returned. Mnemonics for these negative indices as listed here are defined in the <**dial.h**> header file.

```
INTRPT    - 1    /* interrupt occured */
D_HUNG    - 2    /* dialer hung (no return from write) */
NO_ANS    - 3    /* no answer within 10 seconds */
ILL_BD    - 4    /* illegal baud-rate */
A_PROB    - 5    /* acu problem (open() failure) */
L_PROB    - 6    /* line problem (open() failure) */
NO_Ldv    - 7    /* can't open LDEVS file */
DV_NT_A   - 8    /* requested device not available */
DV_NT_K   - 9    /* requested device not known */
NO_BD_A   - 10   /* no device available at requested baud */
NO_BD_K   - 11   /* no device known at requested baud */
```

WARNINGS

Including the <**dial.h**> header file automatically includes the <**termio.h**> header file.

Because the above routine uses <**stdio.h**>, the size of programs not otherwise using standard I/O is increased more than might be expected.

BUGS

An *alarm*(2) system call for 3,600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program is to run for an hour or more, error returns from *read*s should be checked for (**errno**==EINTR), and the *read* possibly reissued.

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir – flexible length directory operations

SYNOPSIS

```
#include <dir.h>

DIR *opendir(filename)
char *filename;

struct direct *readdir(dirp)
DIR *dirp;

long telldir(dirp)
DIR *dirp;

seekdir(dirp, loc)
DIR *dirp;
long loc;

rewinddir(dirp)
DIR *dirp;

closedir(dirp)
DIR *dirp;

cc ... -lndir
```

DESCRIPTION

The purpose of this library is to simulate the new flexible length directory names of 4.2bsd Unix on top of the old directory structure of 4.1bsd. It allows programs to be converted immediately to the new directory access interface, so that they need only be relinked when 4.2bsd becomes available.

opendir opens the directory named by *filename* and associates a *directory stream* with it. *opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory.

readdir returns a pointer to the next directory entry. It returns **NULL** upon reaching the end of the directory or detecting an invalid *seekdir* operation.

telldir returns the current location associated with the named *directory stream.*

seekdir sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation was performed. Values returned by *telldir* are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the *telldir* value may be invalidated due to undetected directory compaction. It is safe to use a previous *telldir* value immediately after a call to *opendir* and before any calls to *readdir.*

rewinddir resets the position of the named *directory stream* to the beginning of the directory.

closedir causes the named *directory stream* to be closed, and the structure associated with the DIR pointer to be freed.

See /usr/include/dir.h for a description of the fields available in a directory entry. The preferred way to search the current directory for entry "name" is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dir))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(dirp);
```

```
                        return FOUND;
                }
        closedir(dirp);
        return NOT_FOUND;
```

LINKING

This library is accessed by specifying "-lndir" as the last argument to the compile line, e.g.:

```
cc -o prog prog.c -lndir
```

SEE ALSO

/usr/include/dir.h, open(2), close(2), read(2), lseek(2)

AUTHOR

Kirk McKusick. Report problems to mckusick@ berkeley or ucbvax!mckusick.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value a and the addend value c are given by

$$a = 5\text{DEECE66D}_{16} = 273673163155_8$$
$$c = \text{B}_{16} = 13_8.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48* sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48,* and a pointer to this buffer is the value returned by *seed48.* This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier a, and *param[6]* specifies the 16-bit addend c. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

NOTES

The versions of these routines for the VAX-11 and PDP-11 are coded in assembly language for maximum speed. It requires approximately 80 μsec on a VAX-11/780 and 130 μsec on a PDP-11/70 to generate one pseudo-random number. On other computers, the routines are coded in portable C. The source code for the portable version can even be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* do not exist; instead, they are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval $[0, m-1]$.

SEE ALSO

rand(3C).

NAME

ecvt, fcvt, gcvt – convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to this string. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero; otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by *ndigit*.

Gcvt converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in Fortran F-format, ready for printing; E-format is produced when F-format is not possible. A minus sign, if there is one, or a decimal point is included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**–p**) option of *cc*(1), and others. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

SEE ALSO

brk(2), malloc(3C).

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;

DESCRIPTION

Erf returns the error function of x, defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Erfc, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large x and the result subtracted from 1.0 (e.g. for $x = 5$, 12 places are lost).

SEE ALSO

exp(3M).

NAME

exp, dexp, cexp – Fortran exponential intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = exp(r1)

dp2 = dexp(dp1)
dp2 = exp(dp1)

cx2 = clog(cx1)
cx2 = exp(cx1)
```

DESCRIPTION

Exp returns the real exponential function e^z of its real argument. *Dexp* returns the double-precision exponential function of its double-precision argument. *Cexp* returns the complex exponential function of its complex argument. The generic function *exp* becomes a call to *dexp* or *cexp*, as required, depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x ;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

DESCRIPTION

Exp returns e^x.

Log returns the natural logarithm of x. The value of x must be positive.

Log10 returns the logarithm base ten of x. The value of x must be positive.

Pow returns x^y. The values of x and y may not both be zero. If x is non-positive, y must be an integer.

Sqrt returns the square root of x. The value of x may not be negative.

DIAGNOSTICS

Exp returns **HUGE** when the correct value would overflow, and sets *errno* to ERANGE.

Log and *log10* return 0 and set *errno* to **EDOM** when x is non-positive. An error message is printed on the standard error output.

Pow returns 0 and sets *errno* to **EDOM** when x is non-positive and y is not an integer, or when x and y are both zero. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow, *pow* returns **HUGE** and sets *errno* to **ERANGE.**

Sqrt returns 0 and sets *errno* to **EDOM** when x is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out and the *stream* to be closed.

Fclose is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

DIAGNOSTICS

These functions return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

DESCRIPTION

Feof returns non-zero when EOF has previously been detected reading the named input *stream*; otherwise, it returns zero.

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*; otherwise, it returns zero.

Clearerr resets the error indicator and EOF indicator to zero on the named *stream*.

Fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, ceil, fmod, fabs – floor, ceiling, remainder, absolute value functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Floor returns the largest integer (as a double-precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns x if y is zero; otherwise, it returns the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

Fabs returns $|x|$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (filename, type )
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by *filename* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

Filename points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

r	open for reading
w	truncate or create for writing
a	append; open for writing at end of file, or create for writing
r+	open for update (reading and writing)
w+	truncate or create for update
a+	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

Freopen is typically used to attach the preopened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor by formatting a file structure from the file descriptor. Thus, *fdopen* can be used to access the file descriptors returned by *open*(2), *dup*(2), *creat*(2), or *pipe*(2). (These calls open files but do not return pointers to a FILE structure.) The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer on failure.

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Fread copies *nitems* items of data from the named input *stream* into an array beginning at *ptr*. An item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream* or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The variable *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char* it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Fread and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp ;

double modf (value, iptr)
double value, *iptr;

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq |x| < 1.0$, and the "exponent" n is an integer. *Frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*.

Ldexp returns the quantity $value * 2^{exp}$.

Modf returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, HUGE is returned and *errno* is set to ERANGE.

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, when the value of *ptrname* is 0, 1, or 2, respectively.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

Ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks; otherwise it returns zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal or on a file opened via *popen*(3S).

WARNING

On an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset; however, portability to systems other than requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

DESCRIPTION

Ftw recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure (see *stat*(2)) containing information about the object, and an integer. Possible values of the integer, defined in the <**ftw.h**> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not be executed successfully. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* is a file in a directory with read permission but not execute (search) permission.

Ftw visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or an error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns – 1, and sets the error type in *errno*.

Ftw uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *Ftw* runs more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

Ftw could be made to run faster and use less storage on deep structures at the cost of considerable complexity.

Ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

NAME

int, ifix, idint, real, float, sngl, dble, cmplx, dcmplx, ichar, char – explicit Fortran type conversion

SYNOPSIS

```
integer i, j
real r, s
double precision dp, dq
complex cx
double complex dcx
character *1 ch

i = int(r)
i = int(dp)
i = int(cx)
i = int(dcx)
i = ifix(r)
i = idint(dp)

r = real(i)
r = real(dp)
r = real(cx)
r = real(dcx)
r = float(i)
r = sngl(dp)

dp = dble(i)
dp = dble(r)
dp = dble(cx)
dp = dble(dcx)

cx = cmplx(i)
cx = cmplx(i, j)
cx = cmplx(r)
cx = cmplx(r, s)
cx = cmplx(dp)
cx = cmplx(dp, dq)
cx = cmplx(dcx)

dcx = dcmplx(i)
dcx = dcmplx(i, j)
dcx = dcmplx(r)
dcx = dcmplx(r, s)
dcx = dcmplx(dp)
dcx = dcmplx(dp, dq)
dcx = dcmplx(cx)

i = ichar(ch)
ch = char(i)
```

DESCRIPTION

These functions perform conversion from one data type to another.

Int converts to *integer* form its *real, double precision, complex,* or *double complex* argument. If the argument is *real* or *double precision,* **int** returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (i.e., truncation). For complex types, the above rule is applied to the real part. **Ifix** and **idint** convert only *real* and *double precision* arguments respectively.

Real converts to *real* form an *integer, double precision, complex,* or *double complex* argument. If the argument is *double precision* or *double complex,* as much precision is kept as is possible. If the argument is one of the complex types, the real part is returned. **Float** and **sngl** convert only *integer* and *double precision* arguments, respectively.

Dble converts any *integer, real, complex,* or *double complex* argument to *double precision* form. If the argument is of a complex type, the real part is returned.

Cmplx converts its *integer, real, double precision,* or *double complex* argument(s) to *complex* form.

Dcmplx converts its *integer, real, double precision,* or *complex* argument(s) to *double complex* form.

Either one or two arguments may be supplied to **cmplx** and **dcmplx** . If there is only one argument, it is taken as the real part of the complex type and a imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part.

Ichar converts from a character to an integer depending on the character's position in the collating sequence.

Char returns the character in the *i*th position in the processor collating sequence, where *i* is the supplied argument.

For a processor capable of representing *n* characters,

ichar(**char**(i)) = i for 0 <= i < *n*, and

char(**ichar**(ch)) = ch for any representable character *ch*.

NAME

gamma – log gamma function

SYNOPSIS

```
#include <math.h>

extern int signgam;

double gamma (x)
double x;
```

DESCRIPTION

Gamma returns the natural log of gamma as a function of the absolute value of a given value. *Gamma* returns ln($|\Gamma(x)|$), where $\Gamma(x)$ is defined as

$$\int_0^\infty e^{-t} t^{x-1} dt.$$

The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LOGHUGE)
        error( );
y = signgam * exp(y);
```

where LOGHUGE is the least value that causes *exp*(3M) to return a range error.

DIAGNOSTICS

For non-negative integer arguments HUGE is returned, and *errno* is set to EDOM. A message indicating DOMAIN error is printed on the standard error output.

If the correct value would overflow, *gamma* returns HUGE and sets *errno* to ERANGE.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

exp(3M), matherr(3M).

NAME

getarg – return Fortran command-line argument

SYNOPSIS

character*N c
integer i

getarg (i, c)

DESCRIPTION

Getarg returns the *i*-th command-line argument of the current process. Thus, if a program were invoked via

foo *arg1 arg2 arg3*

getarg(2, c) would return the string *arg2* in the character variable *c*.

SEE ALSO

getopt(3C).

NAME

getc, getchar, fgetc, getw – get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream*. It also moves the file pointer, if defined, ahead one character in *stream*. *Getc* is a macro and therefore cannot be used if a function is necessary; for example, one cannot have a function pointer point to it.

Getchar returns the next character from the standard input stream, *stdin*. As in the case of *getc*, *getchar* is a macro.

Fgetc performs the same function as *getc*, but is a genuine function. *Fgetc* runs more slowly than *getc*, but takes less space per invocation.

Getw returns the next word (i.e., integer) from the named input *stream*. The size of a word varies from machine to machine. It returns the constant **EOF** upon end-of-file or error, but as that is a valid integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *Getw* increments the associated file pointer, if defined, to point to the next word. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the integer constant **EOF** at end-of-file or upon an error.

BUGS

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, **getc(*f++)** doesn't work sensibly. *Fgetc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

getcwd – get pathname of current working directory

SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

DESCRIPTION

Getcwd returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, *getcwd* obtains *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

Returns NULL with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present; otherwise a NULL pointer is returned.

SEE ALSO

environ(5).

NAME

getenv – return Fortran environment variable

SYNOPSIS

character ***N** c

getenv(TMPDIR, c)

DESCRIPTION

Getenv returns the character-string value of the environment variable represented by its first argument into the character variable of its second argument. If no such environment variable exists, all blanks are returned.

SEE ALSO

getenv(3C), environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent – obtain group file entry from a group file

SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )
```

DESCRIPTION

Getgrent, *getgrgid*, and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a group structure, defined in the **<grp.h>** header file.

```
struct group {
        char    *gr_name;   /* the name of the group */
        char    *gr_passwd; /* the encrypted group password */
        int     gr_gid;     /* the numerical group ID */
        char    **gr_mem;   /* vector of pointers to member names */
};
```

When first called, *getgrent* returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; therefore, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found; it returns a pointer to the particular structure in which the match was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found; it returns a pointer to the particular structure in which the match was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use **<stdio.h>**. This causes them to increase the size of programs not otherwise using standard I/O more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

Getlogin returns a pointer to the login name as found in /etc/utmp. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call *cuserid* or *getlogin*. If *getlogin* fails, call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Getlogin returns the NULL pointer if *name* is not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns EOF. The special option – – may be used to delimit the end of the options; EOF will be returned, and – – will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*.

WARNING

The above routine uses **<stdio.h>**. This causes the size of programs not otherwise using standard I/O to increase more than might be expected.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
```

```
                        bproc( );
                    break;
                case 'f':
                    ifile = optarg;
                    break;
                case 'o':
                    ofile = optarg;
                    bufsiza = 512;
                    break;
                case '?':
                    errflg++;
                }
            if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (2);
            }
            for ( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
            .
            .
            .
        }
```

SEE ALSO

getopt(1).

NAME

getpass – read a password

SYNOPSIS

char ***getpass** (prompt)
char *prompt;

DESCRIPTION

Getpass reads up to a newline or EOF from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echo. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a NULL pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses <**stdio.h**>. This causes the size of programs not otherwise using standard I/O to increase more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw – get name from UID

SYNOPSIS

int getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

Getpw searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *Getpw* returns non-zero if *uid* cannot be found.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNING

The above routine uses <**stdio.h**>. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )
```

DESCRIPTION

Getpwent, *getpwuid*, and *getpwnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a *passwd* structure, declared in the **<pwd.h>** header file:

```
struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};

struct comment {
        char    *c_dept;
        char    *c_name;
        char    *c_acct;
        char    *c_bin;
};
```

Because this structure is declared in **<pwd.h>**, it is not necessary to redeclare it.

The *pw_comment* field is unused; the others have meanings described in *passwd*(4).

When first called, *getpwent* returns a pointer to the first *passwd* structure in the file; thereafter, it returns a pointer to the next *passwd* structure in the file; therefore, successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found; it returns a pointer to the particular structure in which the match was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found; it returns a pointer to the particular structure in which the match was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

The above routines use <**stdio.h**>. Therefore the size of programs not otherwise using standard I/O is increased more than might be expected.

BUGS

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s* until *n*– 1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error (e.g., trying to use these functions on a file that has not been opened for reading) occurs, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file )
char *file;
```

DESCRIPTION

Getutent, *getutid*, and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char    ut_user[8];         /* User login name */
        char    ut_id[4];           /* /etc/inittab id (usually line #) */
        char    ut_line[12];        /* device name (console, lnxx) */
        short   ut_pid;             /* process id */
        short   ut_type;            /* type of entry */
        struct  exit_status {
          short     e_termination;  /* Process termination status */
          short     e_exit;         /* Process exit status */
        } ut_exit;                  /* The exit status of a process
                                    /* marked as DEAD_PROCESS. */
        time_t  ut_time;            /* time entry was made */
};
```

Getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

Getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id- >ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id- >ut_id*. *Getutid* fails if the end of file is reached without a match.

Getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line- >ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is assumed that the user of *pututline* has searched for the proper entry using one of the *getut* routines. If this has been done, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

Setutent resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

Endutent closes the currently open file.

Utmpname allows the user to change the name of the file examined from **/etc/utmp** to any other filename. It is expected that most often this other file will be **/etc/wtmp**. If the file doesn't exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file, if it is currently open, and saves the new filename.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A **NULL** pointer is returned upon failure to read or write. Failure to read may be due to permissions or because end-of-file has been reached.

COMMENTS

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the search of the static structure results in a match, no further search is performed. To use *getutline* to search for multiple occurences, zero out the static structure after each success; otherwise *getutline* will just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. If the implicit read done by *pututline* finds that it isn't already at the correct place in the file, the contents of the static structure returned by the *getutent*, *getutid*, or *getutline* routines are not harmed, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

host - host library

SYNOPSIS

#include <host.h>
cc ... -lhost

DESCRIPTION

The host libary contains a number of routines for easily accessing information in the binary host table. The basic routines return pointers to in-core host structures or the constant BADHOST in case of error. The in core host structure looks like:

```
struct host
{
        short   h_capa;         /* Capability bits */
        long *  h_addrs;        /* Array of addresses (ending with 0) */
        char ** h_names;        /* Array of name string pointers (ending with 0) */
        char *  h_system;       /* Operating system name */
        char ** h_protos;       /* Array of protocol string pointers (ending with 0) */
        char *  h_machine;      /* System hardware type string */
};
```

Where the bits of *h_capa* have the following meaning:

Bit	Defined Name	Meaning
01	NET	Entry is a network definition
02	GATEWAY	Entry is a gateway
04	HOST	Entry is a host
010	SERVER	Entry can be a server
020	USER	Entry is a user

struct host * **addr_info** *(addr)*
long *addr* ;

Returns a pointer to a host entry describing the host or network with internet address *addr.*

struct host * **host_info** *(name)*
char * *name* ;

Returns a pointer to a host entry describing the host or network with name *name.*

struct host * **host_here** ( )

Returns the host structure describing this host.

All of the above routines allocate new space for the returned *host* structure every time they are called. The routine

host_free *(hp)*
struct host * *hp* ;

may be used to free all core space associated with the *host* structure at *hp.*

struct host * **host_next** ( )

returns pointers to successive hosts from the host table. Space allocated by the most recent call to **host_next** ( ) is released during the next call. BADHOST is returned when there are no more hosts. Hosts are returned in the same order as compiled. The function

host_start ( )

resets the internal pointer back to the beginning of the table.

Several routines return string pointers. The pointer NULL is returned upon error. Space is allocated for return strings using **malloc** (), so **free** () should be used to release it when no longer needed.

char * **chaos_name** *(addr)*
short *addr* ;
The primary name of the chaos host at the 16 bit chaos address *addr* is returned. Only hosts on the same network as this system's host are considered.

char * **chaos_sname** *(addr)*
short *addr* ;
Just like *chaos_name* except that certain prefixes (such as mit-) and suffixes (such as -hub) are stripped.

char * **host_name** *(name)*
char * *name* ;
The primary name of the host named *name* is returned.

char * **host_system** *(name)*
char * *name* ;
The operating system of the host named *name* is returned.

char * **host_machine** *(name)*
char * *name* ;
The hardware type of the host named *name* is returned.

char * **host_me** ()
Returns the name of the current host.

The final class of routines return short or long integers. Except where noted, 0 is returned upon error. No permanent storage is allocated when these routines are called.

int **net_number** *(name)*
char * *name* ;
The primary network number of the network named *name* is returned.

short **chaos_addr** *(name, subnet)*
char * *name* ;
int *subnet* ;
The 16 bit address of the chaos host on this host's network named *name* is returned. If the host has several chaos addresses, the one who's high order address byte is equal to *subnet* is preferred over others.

short **arpa_addr** *(name)*
char * *name* ;
The 16 bit arpanet address ((host << 8) |imp) of a host named *name* is returned.

ip_addr *(name, net, subnet, ip)*
char * *name* ;
int *net, subnet* ;
long * *ip* ;
The 32 bit internet address of a host named *name* is stuffed into *ip*. *Net* and *subnet* are the prefered network and arpa host numbers, used to select addresses in hosts which have many. Zero is returned if the search was successful, 1 if it failed.

int **host_hash** *(ptr, len, range)*
A hash value from 0 to *range* - 1 is returned for the item at *ptr* of *len* bytes. **addr_info** () and **host_info** () use **host_hash** () as described in hostbin(5).

FILES

/usr/lib/libhost.a
/usr/include/host.h
/etc/hostbin
/etc/myhostname

SEE ALSO

newhosts(1), hostbin(5), myhostname(5)

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

DESCRIPTION

Hsearch is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the **<search.h>** header file) containing two pointers. *Item.key* points to the comparison key and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION, indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

Hcreate allocates sufficient space for the table and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

Hdestroy destroys the search table and may be followed by another call to *hcreate*.

NOTES

Hsearch uses *open addressing* with a *multiplicative* hash function. However, many other options are available in the source code. The user may select an option by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* (see *string*(3C)).

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available.

- **START** Place new entries at the beginning of the linked list (default is at the end).
- **SORTUP** Keep the linked list sorted by key in ascending order.
- **SORTDOWN** Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining a debugging printout (– DDEBUG) and for including a test driver in the calling routine (– DDRIVER). The source code should be consulted for further details.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C), tsearch(3C).

DIAGNOSTICS

Hsearch returns a NULL pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full.

Hcreate returns zero if it cannot allocate sufficient space for the table.

BUGS

Only one hash search table may be active at any given time.

NAME

hypot – Euclidean distance function

SYNOPSIS

#include <math.h>

double hypot (x, y)
double x, y;

DESCRIPTION

Hypot returns the following, taking precautions against unwarranted overflows:

sqrt(x * x + y * y)

DIAGNOSTICS

When the correct value would overflow, *hypot* returns HUGE and sets *errno* to ERANGE.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M), sqrt(3F).

NAME

index – return location of Fortran substring

SYNOPSIS

```
character *N1 ch1
character *N2 ch2
integer i
i = index(ch1, ch2)
```

DESCRIPTION

Index returns the location of substring *ch2* in string *ch1*. The value returned is either the position at which substring *ch2* starts or 0 if *ch2* is not present in string *ch1*.

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

L3tol converts a list of *n* 3-byte integers (packed into a character string pointed to by *cp*) into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

SEE ALSO

fs(4).

BUGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If TYPE(*ldptr*) is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

Ldahread returns SUCCESS or FAILURE. *Ldahread* fails if TYPE(*ldptr*) does not represent an archive file or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldclose, ldaclose - close a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldopen(3X) and *ldclose* are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If TYPE(*ldptr*) does not represent an archive file, *ldclose* closes the file and frees the memory allocated to the LDFILE structure associated with *ldptr*. If TYPE(*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, *ldclose* reinitializes OFFSET(*ldptr*) to the file address of the next archive member and returns FAILURE. The LDFILE structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns SUCCESS.

Ldaclose closes the file and frees the memory allocated to the LDFILE structure associated with *ldptr* regardless of the value of TYPE*(ldptr)*. *Ldaclose* always returns SUCCESS. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread - read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

Ldfhread returns SUCCESS or FAILURE. *Ldfhread* fails if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro HEADER(*ldptr*) defined in <ldfcn.h> (see *ldfcn*(4)). The information in any field, *fieldname*, of the file header may be accessed using HEADER(*ldptr*).*fieldname*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetname – retrieve symbol name for object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char ldgetname (ldptr, symbol)
LDFILE ldptr;
SYMENT symbol;
```

DESCRIPTION

Ldgetname returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer local to *ldgetname*. Because the buffer is overwritten by each call to *ldgetname*, it must be copied by the caller if the name is to be saved.

The common object file format has been extended to handle arbitrary length symbol names with the addition of a "string table". *Ldgetname* returns the symbol name associated with a symbol table entry for either an object file or a pre-object file. Thus, *ldgetname* can be used to retrieve names from object files without any backward compatibility problems. *Ldgetname* returns **NULL** (defined in **<stdio.h>**) for an object file if the name cannot be retrieved. This occurs when:

- the string table cannot be found.
- not enough memory can be allocated for the string table.
- the string table appears not to be a string table (e.g., if an auxiliary entry is handed to *ldgetname* that looks like a reference to a name in a non-existent string table).
- the name's offset into the string table is beyond the end of the string table.

Typically, *ldgetname* is called immediately after a successful call to *ldtbread* to retrieve the name associated with the symbol table entry filled by *ldtbread*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO linent;
```

DESCRIPTION

Ldlread searches the line number entries of the common object file currently associated with *ldptr*. *Ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *Ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlread, *ldlinit*, and *ldlitem* each return either SUCCESS or FAILURE. *Ldlread* fails if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *Ldlinit* fails if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *Ldlitem* fails if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnlseek seeks to the line number entries of the section specified by *sectname*.

Ldlseek and *ldnlseek* return SUCCESS or FAILURE. *Ldlseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnlseek* fails if there is no section name corresponding to **sectname*. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

Ldohseek returns SUCCESS or FAILURE. *Ldohseek* fails if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen - open a common object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

Ldopen and *ldclose*(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value NULL, *ldopen* opens *filename*, allocates and initializes the LDFILE structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and TYPE(*ldptr*) is the archive magic number, *ldopen* reinitializes the LDFILE structure for the next archive member of *filename*.

Ldopen and *ldclose* are designed to work in concert. *Ldclose* returns FAILURE only when TYPE(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a NULL *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*.

```
/* for each filename to be processed */
ldptr = NULL;
do
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not NULL, *ldaopen* opens *filename* anew and allocates and initializes a new LDFILE structure, copying the TYPE, OFFSET, and HEADER fields from *oldptr*. *Ldaopen* returns a pointer to the new LDFILE structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return NULL if *filename* cannot be opened or if memory for the LDFILE structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnrseek seeks to the relocation entries of the section specified by *sectname*.

Ldrseek and *ldnrseek* return SUCCESS or FAILURE. *Ldrseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnrseek* fails if there is no section name corresponding with *sectname*. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **liblld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread - read an indexed/named section header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char sectname;
SCNHDR *secthead;
```

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

Ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

Ldshread and *ldnshread* return SUCCESS or FAILURE. *Ldshread* fails if *sectindx* is greater than the number of sections in the object file; *ldnshread* fails if there is no section name corresponding with *sectname*. Either function fails if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnsseek seeks to the section specified by *sectname*.

Ldsseek and *ldnsseek* return SUCCESS or FAILURE. *Ldsseek* fails if *sectindx* is greater than the number of sections in the object file; *ldnsseek* fails if there is no section name corresponding with *sectname*. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it returns the the index of the next entry.

Ldtbindex fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

Ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

Ldtbread returns SUCCESS or FAILURE. *Ldtbread* fails if *symindex* is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldgetname(3X), ldopen(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbseek - seek to the symbol table of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek returns SUCCESS or FAILURE. *Ldtbseek* fails if the symbol table has been stripped from the object file or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

len - return length of Fortran string

SYNOPSIS

character*N ch
integer i
i = **len**(ch)

DESCRIPTION

Len returns the length of string *ch*.

NAME

log, alog, dlog, clog – Fortran natural logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = **alog**(r1)
r2 = **log**(r1)

dp2 = **dlog**(dp1)
dp2 = **log**(dp1)

cx2 = **clog**(cx1)
cx2 = **log**(cx1)

DESCRIPTION

Alog returns the real natural logarithm of its real argument. *Dlog* returns the double-precision natural logarithm of its double-precision argument. *Clog* returns the complex logarithm of its complex argument. The generic function *log* becomes a call to *alog*, *dlog*, or *clog* depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

log10, alog10, dlog10 – Fortran common logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **alog10**(r1)
r2 = **log10**(r1)

dp2 = **dlog10**(dp1)
dp2 = **log10**(dp1)

DESCRIPTION

Alog10 returns the real common logarithm of its real argument. *Dlog* returns the double-precision common logarithm of its double-precision argument. The generic function *log* becomes a call to *alog* or *dlog* depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the $LOGNAME variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

BUGS

The return values point to static data whose content is overwritten by each call.

This method of determining a login name is subject to forgery.

NAME

lsearch – linear search and update

SYNOPSIS

```
char *lsearch ((char *)key,(char *)base, nelp, sizeof(*key), compar
unsigned *nelp;
int (*compar)( );
```

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where data may be found. If the data does not occur, it is added at the end of the table. *Key* points to the data to be sought in the table. *Base* points to the first element in the table. *Nelp* points to an integer containing the current number of elements in the table. The integer is incremented if the data is added to the table. *Compar* is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

BUGS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first contiguous reach of free space of sufficient size found in a circular search from the last block allocated or freed; it coalesces adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, *realloc* asks *malloc* to enlarge the arena by *size* bytes and then moves the data to the new space.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc*, and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, *realloc*, and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTE

Search time increases when many objects have been allocated; i.e., if a program allocates space but never frees it, each successive allocation takes longer.

NAME

matherr – error-handling function

SYNOPSIS

#include <math.h>

int matherr (x)
struct exception *x;

DESCRIPTION

Matherr is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors by including a function named *matherr* in their programs. *Matherr* must be of the form described above. A pointer to the exception structure *x* will be passed to the user-supplied *matherr* function when an error occurs. This structure, which is defined in the **<math.h>** header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred; one of the following constants (defined in the header file) is used:

DOMAIN	domain error
SING	singularity
OVERFLOW	overflow
UNDERFLOW	underflow
TLOSS	total loss of significance
PLOSS	partial loss of significance

The element *name* points to a string containing the name of the function that had the error. The variables *arg1* and *arg2* are the arguments to the function that had the error. *Retval* is a double that is returned by the function having the error. If it supplies a return value, the user's *matherr* must return non-zero. If the default error value is to be returned, the user's *matherr* must return 0.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are summarized in the table following the example below. In every case, *errno* is set to non-zero and the program continues.

EXAMPLE

```
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
        case SING: /* print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
        case OVERFLOW:
                if (!strcmp("exp", x->name)) {
                        /* if exp, print message, return the argument */
                        fprintf(stderr, "exp of %f\n", x->arg1);
                        x->retval = x->arg1;
                } else if (!strcmp("sinh", x->name)) {
```

```
                        /* if sinh, set errno, return 0 */
                        errno = ERANGE;
                        x->retval = 0;
                } else
                        /* otherwise, return HUGE */
                        x->retval = HUGE;
                break;
        case UNDERFLOW:
                return (0); /* execute default procedure */
        case TLOSS:
        case PLOSS:
                /* print message and return 0 */
                fprintf(stderr, "loss of significance in %s\n", x->name);
                x->retval = 0;
                break;
        }
        return (1);
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
BESSEL: y0, y1, yn (neg. no.)	– M, – H	– –	H –	0 –	– –	* –
EXP:	–	–	H	0	–	
POW: (neg.)**(non-int.), 0**0	– M, 0	– –	H –	0 –	– –	– –
LOG: log(0): log(neg.):	 – M, – H	 M, – H –	 – –	 – –	 – –	 – –
SQRT:	M, 0	–	–	–	–	–
GAMMA:	–	M, H	–	–	–	–
HYPOT:	–	–	H	–	–	–
SINH, COSH:	–	–	H	–	–	–
SIN, COS:	–	–	–	–	M, 0	M, *
TAN:	–	–	H	–	0	*
ACOS, ASIN:	M, 0	–	–	–	–	–

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed.
H	HUGE is returned.
– H	– HUGE is returned.
0	0 is returned.

NAME

max, max0, amax0, max1, amax1, dmax1 – Fortran maximum-value functions

SYNOPSIS

integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3

l = **max**(i, j, k)
c = **max**(a, b)
dp = **max**(a, b, c)
k = **max0**(i, j)
a = **amax0**(i, j, k)
i = **max1**(a, b)
d = **amax1**(a, b, c)
dp3 = **dmax1**(dp1, dp2)

DESCRIPTION

The maximum-value functions return the largest of their arguments; there may be any number of arguments. *Max* is the generic form which can be used for all data types and takes its return type from that of its arguments. All arguments must be of the same type. *Max0* returns the integer form of the maximum value of its integer arguments; *amax0*, the real form of its integer arguments; *max1*, the integer form of its real arguments; *amax1*, the real form of its real arguments; and *dmax1*, the double-precision form of its double-precision arguments.

SEE ALSO

min(3F).

NAME

mclock – return Fortran time accounting

SYNOPSIS

integer i

i = **mclock( )**

DESCRIPTION

Mclock returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

times(2), clock(3C), system(3F).

NAME

memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied or after *n* characters have been copied, whichever comes first. It returns either a pointer to the character after the copy of *c* in *s1* or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

Memchr returns either a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s* or a NULL pointer if *c* does not occur.

Memcmp compares its arguments, looking at the first *n* characters only. It returns an integer less than, equal to, or greater than 0, depending on whether *s1* is lexicographically less than, equal to, or greater than *s2*.

Memcpy copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

Memset sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

NOTE

For user convenience, all these functions are declared in the optional **<memory.h>** header file.

BUGS

Memcmp uses native character comparison, which is unsigned on M68010 and may be signed on other machines.

Because character movement is performed differently in different implementations, overlapping moves may yield unexpected results.

NAME

min, min0, amin0, min1, amin1, dmin1 – Fortran minimum-value functions

SYNOPSIS

integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3

l = **min**(i, j, k)
c = **min**(a, b)
dp = **min**(a, b, c)
k = **min0**(i, j)
a = **amin0**(i, j, k)
i = **min1**(a, b)
d = **amin1**(a, b, c)
dp3 = **dmin1**(dp1, dp2)

DESCRIPTION

The minimum-value functions return the minimum of their arguments. There may be any number of arguments. *Min* is the generic form which can be used for all data types. It takes its return type from that of its arguments, which must all be of the same type. *Min0* returns the integer form of the minimum value of its integer arguments; *amin0*, the real form of its integer arguments; *min1*, the integer form of its real arguments; *amin1*, the real form of its real arguments; and *dmin1*, the double-precision form of its double-precision arguments.

SEE ALSO

max(3F).

NAME

mktemp – make a unique filename

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* with a unique filename; it returns the address of *template*. The string in *template* should look like a filename with six trailing **X**s; *mktemp* replaces the **X**s with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

BUGS

It is possible to run out of letters.

NAME

mod, amod, dmod – Fortran remaindering intrinsic functions

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3

k = mod(i, j)

r3 = amod(r1, r2)
r3 = mod(r1, r2)

dp3 = dmod(dp1, dp2)
dp3 = mod(dp1, dp2)
```

DESCRIPTION

Mod returns the integer remainder of its first argument divided by its second argument. *Amod* and *dmod* return, respectively, the real and double-precision whole number remainder of the integer division of their two arguments. The generic version *mod* returns the data type of its arguments.

NAME

monitor – prepare execution profile

SYNOPSIS

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short *buffer;
int bufsize, nfunc;

DESCRIPTION

An executable program created by **cc – p** automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. *Monitor* arranges to record a histogram in the buffer. This histogram shows periodically sampled values of the program counter and counts of calls of certain functions. The lowest address sampled is that of *lowpc*; the highest address is just below *highpc*. *Lowpc* may not equal 0 for this use of *monitor*. *Nfunc* is the maximum number of call counts that can be kept; only calls of functions compiled with the profiling option – **p** of *cc*(1) are recorded. (The C Library and Math Library supplied when **cc -p** is used also have call counts recorded.) For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor ((int (*)())2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text; see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor ((int (*)())NULL, 0, 0, 0, 0);
```

Prof(1) can then be used to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

NAME

nlist - get entries from name list

SYNOPSIS

#include <a.out.h>

int nlist (filename, nl)
char *filename;
struct nlist nl[ ];

DESCRIPTION

Nlist examines the name list in the executable file whose name is pointed to by *filename*; it selectively extracts a list of values and puts them in the array of *nlist* structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types, and values. The list is terminated with a null name; i.e., a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(4) for a discussion of the symbol table structure.

This subroutine is useful for examining the system name list kept in the file **/unix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(4).

DIAGNOSTICS

All type entries are set to 0 if the file cannot be read or if it doesn't contain a valid name list.

Nlist returns – 1 upon error; otherwise it returns 0.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

plot – graphics interface subroutines

SYNOPSIS

```
openpl ()

erase ()

label (s)
char *s;

line (x1, y1, x2, y2)
int x1, y1, x2, y2;

circle (x, y, r)
int x, y, r;

arc (x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

move (x, y)
int x, y;

cont (x, y)
int x, y;

point (x, y)
int x, y;

linemod (s)
char *s;

space (x0, y0, x1, y1)
int x0, y0, x1, y1;

closepl ()
```

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. *Space* must be used before any of these functions to declare the amount of space necessary; see *plot*(4). *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

Circle draws a circle of radius *r* with center at the point *(x, y)*.

Arc draws an arc of a circle with center at the point *(x, y)* between the points *(x0, y0)* and *(x1, y1)*.

String arguments to *label* and *linemod* are terminated by nulls and do not contain new-lines.

See *plot*(4) for a description of the effect of the remaining functions.

The library files listed below provide several variations of these routines.

FILES

/usr/lib/libplot.a	produces output for *tplot*(1G) filters
/usr/lib/lib300.a	for DASI 300
/usr/lib/lib300s.a	for DASI 300s
/usr/lib/lib450.a	for DASI 450
/usr/lib/lib4014.a	for Tektronix 4014

WARNINGS

To compile a program containing these functions in *file.c*, use **cc***file.c*– **lplot**

To execute it, use **a.out | tplot.**

The above routines use **<stdio.h>**. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

SEE ALSO

graph(1G), stat(1G), tplot(1G), plot(4).

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings; one string contains a shell command line and the other contains an I/O mode. The mode may be either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling program and the command to be executed. The value returned is a stream pointer. If the I/O mode is **w**, one can write to the standard input of the command by writing to the file *stream*; if the I/O mode is **r**, one can read from the standard output of the command, by reading from the file *stream*.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter and a type **w** as an output filter.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created or if the shell cannot be accessed.

Pclose returns – 1 if *stream* is not associated with a command opened by *popen*.

BUGS

If the original processes and processes opened by *popen* concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g., by using *fflush*; see *fclose*(3S).

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ... )
char *format;

int fprintf (stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ... )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output", followed by the null character (**\0**) in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character %. After the % the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded to the field width on the left (default) or right (if the left-adjustment flag has been given); see below for flag specification.

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The format of the precision is a period (.) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen; therefore, the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or −).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (**X**) conversion, a non-zero result will have **0x** (**0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters ABCDEF for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[−]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double *arg* is converted in the style "[−]d.ddde± dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** is used only if the exponent resulting from the conversion is less than − 4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer *arg* has the value zero, the result is undefined. A *null* arg yields undefined results.

% Print a % no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %2d:%2d", weekday, month, day, hour, min);
```

To print *pi* to 5 decimal places:

```
printf("pi = %5f", 4*atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

int putchar (c)
char c;

int fputc (c, stream)
char c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character *c* onto the output *stream* at the position where the file pointer, if defined, is pointing. *Putchar*(*c*) is defined as *putc*(*c, stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw writes the word (i.e., integer) *w* to the output *stream* at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen*(see *fopen*(3S)) causes it to become buffered or line-buffered. When an output stream is unbuffered information, it is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block; when it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (i.e., as soon as a new-line character is written or terminal input is requested). *Setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant **EOF**. This occurs if the file *stream* is not open for writing or if the output file cannot be grown. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent and may not be read using *getw* on a different processor. For this reason the use of *putw* should be avoided.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwuid* writes a line on the stream *f* which matches the format of **/etc/passwd**.

The **<pwd.h>** header file is described in *getpwent*(3C).

SEE ALSO

getpwent(3C).

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation; otherwise it returns zero.

WARNING

The above routine uses **<stdio.h>**. Therefore, the size of programs not otherwise using standard I/O is increased more than might be expected.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream *stdout*.

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

DIAGNOSTICS

Both routines return EOF on error. This occurs if the routines try to write on a file that has not been opened for writing.

NOTES

Puts appends a new-line character while *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar
unsigned int nel;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Base points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. Depending on whether the first argument is to be considered less than, equal to, or greater than the second argument, the *compar* function must return an integer less than, equal to, or greater than zero.

NOTES

The pointer to the base of the table should be of type pointer-to-element and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

Srand can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

NAME

srand, rand – Fortran uniform random-number generator

SYNOPSIS

```
integer i, j
call srand(i)
j = rand( )
```

DESCRIPTION

Srand takes its integer argument as the seed of a random-number generator, the values of which are returned through successive invocations of *rand*.

SEE ALSO

rand(3C).

NAME

regcmp, regex – compile and execute a regular expression

SYNOPSIS

char *regcmp(string1 [, string2, ...], 0)
char *string1, *string2, ...;

char *regex(re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *loc1;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space that has been allocated by *malloc*. A NULL return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^ — These symbols retain their current meaning.

$ — This symbol matches the end of the string; \n matches the new-line.

– — Within brackets the minus means "through". For example, [a– z] is equivalent to [abcd...xyz]. The – can appear as itself only if used as the last or first character. For example, the character class expression []–] matches the characters] and – .

\+ — A regular expression followed by + means "one or more times". For example, [0– 9]+ is equivalent to [0– 9][0– 9]*.

{*m*} {*m*,} {*m*,*u*} — Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The minimum number is *m* and the maximum number is *u*, which must be less than 256. If only *m* is present (e.g., {*m*}), it indicates the exact number of times the regular expression is to be applied. {*m*,} is analogous to {*m*,**infinity**}. The plus (+) and star (*) operations are equivalent to {**1**,} and {**0**,}, respectively.

(...)$*n* — The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+ 1)*th argument following the subject argument. At present, at most 10 enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) — Parentheses are used for grouping. An operator (e.g., *, +, { }) can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
        ...
name = regcmp("([A- Za- z][A- za- z0- 9_]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+ 11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        ...
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) reuses the same vector, saving time and space:

```
/* user's program */
        ...
malloc(n) {
        static int rebuf[256];
        return rebuf;
}
```

NAME

anint, dnint, nint, idnint – Fortran nearest integer functions

SYNOPSIS

```
integer i
real r1, r2
double precision dp1, dp2

r2 = anint(r1)
i = nint(r1)

dp2 = anint(dp1)
dp2 = dnint(dp1)

i = nint(dp1)
i = idnint(dp1)
```

DESCRIPTION

Anint returns the nearest whole real number to its real argument (i.e., int(a+0.5) if a $\geq$ 0, int(a–0.5) otherwise). *Dnint* does the same for its double-precision argment. *Nint* returns the nearest integer to its real argument. *Idnint* is the double-precision version. *Anint* is the generic form of *anint* and *dnint*, performing the same operation and returning the data type of its argument. *Nint* is also the generic form of *idnint*.

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf (format [ , pointer ] ... )
char *format;

int fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;

int sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to *format*, and stores the results in its arguments. Each function expects two arguments: a control string *format* (described below) and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks and tabs) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character % an optional assignment suppression character *, an optional numerical maximum field width, an optional **l** or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression has been indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-white-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument should be given. The following conversion codes are legal:

%	A single %is expected in the input at this point; no assignment is done.
d	A decimal integer is expected; the corresponding argument should be an integer pointer.
u	An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
o	An octal integer is expected; the corresponding argument should be an integer pointer.
x	A hexadecimal integer is expected; the corresponding argument should be an integer pointer.
e,f,g	A floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optionally signed integer.
s	A character string is expected; the corresponding argument should be a character pointer to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a white-space

character.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[String data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters (the *scanset*) and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the *scanset*. The circumflex, (^), when it appears as the first character in the *scanset*, serves as a complement operator and redefines the *scanset* as the set of all characters *not* contained in the remainder of the *scanset* string. There are some conventions used in the construction of the *scanset*. A range of characters may be represented by the construct *first-last*; thus, **[0123456789]** may be expressed **[0-9]**. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the *scanset*. To include the right square bracket as an element of the *scanset*, it must appear as the first character (possibly preceded by a circumflex) of the *scanset*; otherwise it will be interpreted syntactically as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which will be added automatically.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or **short**, rather than **int**, is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double**, rather than **float**, is in the argument list.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero when an early conflict between an input character and the control string occurs. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line

25 54.32E-1 thompson

will assign the value **25** to *i*, and the value **5.432** to *x*; *name* will contain **thompson\0**.

The call

```
int i; float x; char name[50];
scanf ("%2d%f%*d %[0-9]", &i, &x, name);
```

with input

56789 0123 56a72

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

atof(3C), getc(3S), printf(3S), strtol(3C).

NOTE

Trailing white space is left unread unless matched in the control string.

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is a NULL character pointer, input/output will be completely unbuffered.

A constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from *malloc*(3C) at the time of the first *getc*(3S) or *putc*(3S) on the file, except that the standard error stream *stderr* is normally not buffered.

Output streams directed to terminals are always line-buffered unless they are unbuffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S).

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block and then failing to close the stream in the same block.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. The environment type *jmp_buf* is defined in the **<setjmp.h>** header file. *Setjmp* returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2).

WARNING

Longjmp fails if it is called when *env* was never primed by a call to *setjmp* or when the last such call is in a function which has since returned.

NAME

sign, isign, dsign – Fortran transfer-of-sign intrinsic function

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3

k = isign(i, j)
k = sign(i, j)

r3 = sign(r1, r2)

dp3 = dsign(dp1, dp2)
dp3 = sign(dp1, dp2)
```

DESCRIPTION

Isign returns the magnitude of its first argument with the sign of its second argument. *Sign* and *dsign* are its real and double-precision counterparts, respectively. The generic version is *sign*, which devolves to the appropriate type depending on its arguments.

NAME

signal – specify Fortran action on receipt of a system signal

SYNOPSIS

integer i
external integer intfnc

call signal(i, intfnc)

DESCRIPTION

Signal allows a process to specify a function to be invoked upon receipt of a specific signal. The first argument specifies a fault or exception; the second argument specifies the function to be invoked.

SEE ALSO

kill(2), signal(2).

NAME

sin, dsin, csin – Fortran sine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = **sin**(r1)

dp2 = **dsin**(dp1)
dp2 = **sin**(dp1)

cx2 = **csin**(cx1)
cx2 = **sin**(cx1)

DESCRIPTION

Sin returns the real sine of its real argument. *Dsin* returns the double-precision sine of its double-precision argument. *Csin* returns the complex sine of its complex argument. The generic *sin* function becomes *dsin* or *csin* as required by argument type.

SEE ALSO

trig(3M).

NAME

sinh, dsinh - Fortran hyperbolic sine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **sinh**(r1)

dp2 = **dsinh**(dp1)
dp2 = **sinh**(dp1)

DESCRIPTION

Sinh returns the real hyperbolic sine of its real argument. *Dsinh* returns the double-precision hyperbolic sine of its double-precision argument. The generic form *sinh* may be used to return a double-precision value given a double-precision argument.

SEE ALSO

sinh(3M).

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

Sinh, *cosh*, and *tanh* return, respectively, the hyberbolic sine, cosine, and tangent of their argument.

DIAGNOSTICS

Sinh and *cosh* return HUGE when the correct value would overflow and set *errno* to ERANGE.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

Sleep suspends the current process from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) any caught signal will terminate *sleep* following execution of the signal catching routine. The suspension time may be longer than requested by an arbitrary amount, due to the scheduling of other activity in the system. The value returned by *sleep* is the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time or in case there is premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time before the alarm signal, the process sleeps only until the alarm signal would have occurred and the caller's alarm catch routine is executed just before the *sleep* routine returns. If the *sleep* time is less than the time before the calling program's alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sputl, sgetl – access long integer data in a machine independent fashion.

SYNOPSIS

void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;

DESCRIPTION

Sputl takes the 4 bytes of the long integer *value* and places them in memory, starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

Sgetl retrieves the 4 bytes in memory, starting at the address pointed to by *buffer*, and returns the long integer value in the byte ordering of the host machine.

Use of *sputl* and *sgetl* in combination provides a machine independent way of storing long numeric data in a file in binary form without conversion to characters.

A program that uses these functions must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ar(4).

NAME

sqrt, dsqrt, csqrt – Fortran square root intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = **sqrt**(r1)

dp2 = **dsqrt**(dp1)
dp2 = **sqrt**(dp1)

cx2 = **csqrt**(cx1)
cx2 = **sqrt**(cx1)

DESCRIPTION

Sqrt returns the real square root of its real argument. *Dsqrt* returns the double-precision square root of its double-precision arguement. *Csqrt* returns the complex square root of its complex argument. *Sqrt*, the generic form, will become *dsqrt* or *csqrt* as required by its argument type.

SEE ALSO

exp(3M).

NAME

ssignal, gsignal – software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions; it is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to *ssignal* associates a procedure, *action*, with the software signal, *sig*; the software signal, *sig*, is raised by a call to *gsignal*. Raising a software signal causes the action established for that signal to be taken.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a user-defined *action* function or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). *Ssignal* returns the action previously established for that signal type; if no *action* has been established or the signal number (*sig*) is illegal, *ssignal* returns SIG_DFL.

Gsignal raises the signal identified by its argument, *sig*:

If an *action* function has been established for *sig*, then that *action* is reset to SIG_DFL and the *action* function is entered with argument *sig*. *Gsignal* returns the value returned to it by the *action* function.

If the *action* for *sig* is SIG_IGN, *gsignal* returns the value 1 and takes no other action.

If the *action* for *sig* is SIG_DFL, *gsignal* returns the value 0 and takes no other action.

If *sig* has an illegal value or no *action* was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The input/output function may be grouped into the following categories: file access, file status, input, output, miscellaneous. For lists of the functions in each category, refer to the "Libraries" section of the *Programming Guide*. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file.

A constant NULL (0) designates a nonexistent pointer.

An integer constant EOF (– 1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

#include <stdio.h>

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following functions are implemented as macros: *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *clearerr*, and *fileno*. Redeclaration of these names is perilous.

The **<stdio.h>** file is illustrated in the "Libraries" section of the *Programming Guide*.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers cause serious errors, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdipc – standard interprocess communication package

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If a standard is not adhered to, unrelated processes may interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

Ftok returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the pathname of an existing file that is accessible to the process. *Id* is a character that uniquely identifies a project. *Ftok* returns the same key for linked files when called with the same *id*; it returns different keys when called with the same filename but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

Ftok returns **(key_t) – 1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, *ftok* is likely to return a different key than it did the original time it was called.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* appends at most *n* characters. Each function returns a pointer to the null-terminated result.

Strcmp performs a lexicographical comparison of its arguments and returns an integer less than, equal to, or greater than 0, when *s1* is less than, equal to, or greater than *s2*, respectively. *Strncmp* makes the same comparison but looks at a maximum of *n* characters.

Strcpy copies string *s2* to string *s1*, stopping after the null character has been copied. *Strncpy* copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result is not null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

Strlen returns the number of characters in *s*, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and writes a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that on subsequent calls (which must be made with a NULL pointer as the first argument) it works through the string *s1* immediately following that token. This can be continued until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

NOTE

For user convenience, all these functions are declared in the optional <**string.h**> header file.

BUGS

Strcmp and *strncmp* use native character comparison, which is unsigned on M68010 and may be signed on other machines.

Character movement is performed differently in different implementations; therefore, overlapping moves may yield unexpected results.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

DESCRIPTION

Strtol returns as a long integer the value represented by the character string *str*. The string is scanned up to the first character inconsistent with the base. Leading white-space characters (blanks and tabs) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in **ptr*. If no integer can be formed, zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored; a leading **0x** or **0X** is ignored if *base* is 16.

If *base* is zero, the string itself determines the base. After an optional leading sign, a leading zero indicates octal conversion and a leading **0x** or **0X** indicates hexadecimal conversion; otherwise, decimal conversion is used.

Truncation from long to int can take place upon assignment or by an explicit cast.

Atol(*str*) is equivalent to **strtol**(*str*, **(char **)NULL, 10)**.

Atoi(*str*) is equivalent to (*int*) **strtol**(*str*, **(char **)NULL, 10)**.

SEE ALSO

atof(3C), scanf(3S).

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

void swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines with different byte ordering. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive, *swab* uses *nbytes*– 1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

system – issue a shell command from Fortran

SYNOPSIS

character *N c

call system(c)

DESCRIPTION

System causes its character argument to be given to *sh*(1) as input, as if the string had been typed at a terminal. The current process waits until the shell has completed.

SEE ALSO

sh(1), exec(2), system(3S).

NAME

system – issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

DESCRIPTION

System causes *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System forks to create a child process that in turn performs *exec*(2) on */bin/sh* in order to execute *string*. If the fork or exec fails, *system* returns – 1 and sets *errno*.

NAME

tan, dtan – Fortran tangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **tan**(r1)

dp2 = **dtan**(dp1)
dp2 = **tan**(dp1)

DESCRIPTION

Tan returns the real tangent of its real argument. *Dtan* returns the double-precision tangent of its double-precision argument. The generic *tan* function becomes *dtan* as required with a double-precision argument.

SEE ALSO

trig(3M).

NAME

tanh, dtanh – Fortran hyperbolic tangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2

r2 = **tanh**(r1)

dp2 = **dtanh**(dp1)
dp2 = **tanh**(dp1)

DESCRIPTION

Tanh returns the real hyperbolic tangent of its real argument. *Dtanh* returns the double-precision hyperbolic tangent of its double precision argument. The generic form *tanh* may be used to return a double-precision value given a double-precision argument.

SEE ALSO

sinh(3M).

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, *area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int *outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base *termcap*(5). Note that these are low-level routines.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *Tgetent* returns – 1 if it cannot open the **termcap** file, 0 if the terminal name given does not have an entry, and 1 if successful. It looks in the environment for a TERMCAP variable. If a variable is found whose value does not begin with a slash and the terminal type *name* is the same as the environment string TERM, the TERMCAP string is used instead of reading the **termcap** file. If the value does begin with a slash, the string is used as a pathname rather than **/etc/termcap**. This can speed up entry into programs that call *tgetent*. It can also help debug new terminal descriptions or be used to make one for your terminal if you can't write the file **/etc/termcap**.

Tgetnum gets the numeric value of capability *id*, returning – 1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables UP (from the **up** capability) and BC (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs that call *tgoto* should be sure to turn off the XTABS bit(s), since *tgoto* may now output a tab. Note that programs using *termcap* should in general turn off XTABS anyway since some terminals use control-I for other functions, such as nondestructive space.) If a %sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable; *outc* is a routine that is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty (2)*. The external variable PC should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	– ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), termcap(5)

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ( )

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding FILE pointer. The file is automatically deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S).

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;

DESCRIPTION

These functions generate filenames that can safely be used for a temporary file.

Tmpnam always generates a filename using the pathname defined as *P_tmpdir* in the **<stdio.h>** header file. If *s* is NULL, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in **<stdio.h>**; *tmpnam* places its result in that array and returns *s*.

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the pathname of the directory in which the file is to be created. If *dir* is NULL or points to a string which is not a pathname for an appropriate directory, the pathname defined as *P_tmpdir* in the **<stdio.h>** header file is used. If that pathname is not accessible, **/tmp** will be used as a last resort. This entire sequence can be upstaged by providing an environment variable **TMPDIR** in the user's environment, whose value is a pathname for the desired temporary-file directory.

Many applications prefer that names of temporary files contain favorite initial letter sequences. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to 5 characters to be used as the first few characters of the name of the temporary file.

Tempnam uses *malloc*(3C) to get space for the constructed filename and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* (see *malloc*(3C)). If *tempnam* cannot return the expected result for any reason (i.e., *malloc* failed or attempts to find an appropriate directory were unsuccessful), a NULL pointer will be returned.

NOTES

These functions generate a different filename each time they are called.

Files created using these functions and either *fopen*(2) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

BUGS

If called more than 17,576 times in a single process, *tmpnam* and *tempnam* will start recycling previously used names.

Between the time a filename is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam*, *tempnam*, or *mktemp*(3C) and the filenames are chosen carefully to avoid duplication by other means.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double tan (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

DESCRIPTION

Sin, *cos*, and *tan* return, respectively, the sine, cosine, and tangent of their argument, which is in radians.

Asin returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine of x, in the range 0 to π.

Atan returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

Sin, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to ERANGE.

Tan returns HUGE for an argument which is near an odd multiple of $\pi/2$ when the correct value would overflow; it sets *errno* to ERANGE.

Arguments of magnitude greater than 1.0 cause *asin* and *acos* to return 0 and to set *errno* to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

SEE ALSO

matherr(3M).

NAME

tsearch, tdelete, twalk – manage binary search trees

SYNOPSIS

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );
```

DESCRIPTION

Tsearch is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where data may be found. If the data does not occur, it is added at an appropriate point in the tree. *Key* points to the data to be sought in the tree. *Rootp* points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable is set to point to the data at the root of the new tree. *Compar* is the name of the comparison function. It is called with two arguments that point to the elements being compared. If the first argument is to be considered less than, equal to, or greater than the second argument, the function must return an integer less than, equal to, or greater than zero, respectively.

Tdelete deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for *tsearch*. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node or a NULL pointer if the node is not found.

Twalk traverses a binary search tree. *Root* is the root of the tree to be traversed. Any node in a tree may be used as the root for a walk below that node. *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* As defined in the **<search.h>** header file, the value of this data type depends on whether this is the first, second, or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree; the root is level zero.

NOTES

The pointers to the key and the root of the tree should be of type pointer-to-element and cast to type pointer-to-character.

The comparison function need not compare every byte; therefore, arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element. on entry.

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

BUGS

Tsearch fails if the calling function alters the pointer to the root.

WARNING

The *root* argument to *twalk* is one level of indirection less than the *rootp* arguments to *tsearch* and *tdelete*.

DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tsearch* and *tdelete* if *rootp* is NULL

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char ***ttyname** (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

Ttyname returns a pointer to a string containing the null-terminated pathname of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device; otherwise, it returns 0.

FILES

/dev/*

DIAGNOSTICS

Ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

Ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by scanning the file **/etc/inittab** for the name of the terminal device associated with the standard input, the standard output, or the error output (0, 1, or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error is encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

ungetc – push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc* call on that *stream*. *Ungetc* returns *c* and leaves the file *stream* unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If *c* equals EOF, *ungetc* does nothing to the buffer and returns EOF.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

For *ungetc* to perform correctly, a read statement must have been performed prior to the call of the *ungetc* function. *Ungetc* returns EOF if it can't insert the character. If *stream* is *stdin*, *ungetc* allows exactly one character to be pushed back onto the buffer without a previous read statement.

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the header files and file formats used by C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

NAME

a.out – common assembler and link editor output

DESCRIPTION

A.out is the output file from the assembler *as*(1) and the link editor *ld*(1). *A.out* can be executed on the target machine if there were no errors in assembling or linking and no unresolved external references.

The object file format supports user-defined sections and contains extensive information for symbolic software testing. A common object file consists of a file header, an optional aout header, a table of section headers, relocation information, (optional) line numbers, and a symbol table. The order is given below.

File header.
Optional aout header.
Section 1 header.
...
Section n header.
Section 1 data.
...
Section n data.
Section 1 relocation.
...
Section n relocation.
Section 1 line numbers.
...
Section n line numbers.
Symbol table.
String table.

The last four sections (relocation, line numbers, symbol table, and string table) may be missing if the program was linked with the – **s** option of *ld*(1) or if the symbol table and relocation bits were removed by *strip*(1). Also note that if the program was linked without the – **r** option, the relocation information will be absent. The string table exists only if necessary.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized data, the latter actually being initialized to all 0's), and a stack. The text segment begins at location 0 in the core image; the header is not loaded. If the magic number (the first field in the optional aout header) is 407 (octal), it indicates that the text segment is not to be write-protected or shared, so the data segment will be contiguous with the text segment. If the magic number is 410 (octal), the data segment begins at the next segment boundary following the text segment, and the text segment is not writable by the program. If other processes are executing the same **a.out** file, they will share a single text segment.

On the M68010, the stack begins at the end of the process virtual address space (0xffffff) and grows toward lower addresses. The stack is automatically extended as required. The data segment is extended only as requested by the *brk*(2) and *sbrk*(2) system calls.

The value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text involves a reference to an undefined external symbol, the storage class of the symbol-table entry for that word will be marked as an "external symbol", and the section number will be set to 0. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

See *aouthdr*(4), *filehdr*(4), *linenum*(4), *scnhdr*(4), *reloc*(4), and *syms*(4) for descriptions of the individuals parts. Every section created by *as*(1) contains a multiple-of-four number of bytes; directives to *ld*(1) can create a section with an odd number of bytes.

SEE ALSO

as(1), cc(1), ld(1), aouthdr(4), filehdr(4), ldfcn(4), linenum(4), reloc(4), scnhdr(4), syms(4).

NAME

acct – per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by **<sys/acct.h>**, whose contents are:

```
typedef ushort comp_t; /* "floating point" */
                       /* 13-bit fraction, 3-bit exponent  */

struct   acct
{
        char    ac_flag;        /* Accounting flag */
        char    ac_stat;        /* Exit status */
        ushort  ac_uid;
        ushort  ac_gid;
        dev_t   ac_tty;
        time_t  ac_btime;       /* Beginning time */
        comp_t  ac_utime;       /* acctng user time in clock ticks */
        comp_t  ac_stime;       /* acctng system time in clock ticks */
        comp_t  ac_etime;       /* acctng elapsed time in clock ticks */
        comp_t  ac_mem;         /* memory usage in clicks */
        comp_t  ac_io;          /* chars trnsfrd by read/write */
        comp_t  ac_rw;          /* number of block reads/writes */
        char    ac_comm[8]; /* command name */
};

extern  struct  acct            acctbuf;
extern  struct  inode           *acctp;  /* inode of accounting file */

#define AFORK 01                /* has executed fork, but no exec */
#define ASU   02                /* used superuser privileges */
#define ACCTF 0300              /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem/(ac_stime + ac_utime)* can be viewed as an approximation of the mean process size, as modified by text-sharing.

The structure **tacct.h**, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
 *  total accounting (for acct period), also for day
 */

struct tacct {
        uid_t           ta_uid;      /* userid */
        char            ta_name[8];  /* login name */
        float           ta_cpu[2];   /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2]; /* cum kcore-minutes, p/np */
        float           ta_con[2];   /* cum. connect time, p/np, mins */
        float           ta_du;       /* cum. disk usage */
        long            ta_pc;       /* count of processes */
        unsigned short  ta_sc;       /* count of login sessions */
        unsigned short  ta_dc;       /* count of disk samples */
        unsigned short  ta_fee;      /* fee for special services */
};
```

SEE ALSO

acct(1M), acctcom(1), acct(2).

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

aliases – aliases file for sendmail

SYNOPSIS

/usr/lib/aliases

DESCRIPTION

This file describes user id aliases used by */usr/lib/sendmail*. It is formatted as a series of lines of the form

name: name_1, name2, name_3, . . .

The *name* is the name to alias, and the *name_n* are the aliases for that name. Lines beginning with white space are continuation lines. Lines beginning with '#' are comments.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

After aliasing has been done, local and valid recipients who have a ".forward" file in their home directory have messages forwarded to the list of users defined in that file.

This is only the raw data file; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the program *newaliases*(1). A *newaliases* command should be executed each time the aliases file is changed for the change to take effect.

SEE ALSO

newaliases(1), dbm(3X), sendmail(1M)
SENDMAIL Installation and Operation Guide.
SENDMAIL An Internetwork Mail Router.

BUGS

Because of restrictions in *dbm*(3X) a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by "chaining"; that is, make the last name in the alias be a dummy name which is a continuation alias.

NAME

aouthdr - optional aout header

SYNOPSIS

#include <aouthdr.h>

DESCRIPTION

An object file may contain an optional header, following the file header described in *filehdr*(4). Object files that have been completely linked by *ld*(1) contain this header; others do not. The format of the optional header is:

```
typedef struct aouthdr {
        short   magic;          /* magic number */
        short   vstamp;         /* version stamp */
        long    tsize;          /* text size in bytes, padded (.text) */
        long    dsize;          /* initialized data (.data) */
        long    bsize;          /* uninitialized data (.bss) */
        long    entry;          /* entry point */
        long    text_start;     /* base of text used for this file */
        long    data_start;     /* base of data used for this file */
} AOUTHDR;
```

SEE ALSO

a.out(4), filehdr(4).

NAME

ar – common archive file format

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

Each archive begins with the archive magic string.

```
#define ARMAG "!<arch>\n"       /* magic string */
#define SARMAG 8                /* length of magic string */
```

Each archive which contains common object files (see *a.out*(4)) includes an archive symbol table. This symbol table is used by the link editor *ld*(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by *ar*.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define ARFMAG "`\n"        /* header trailer string */

struct  ar_hdr          /* file member header */
{
        char    ar_name[16];    /* '/' terminated file member name */
        char    ar_date[12];    /* file member date */
        char    ar_uid[6];      /* file member user identification */
        char    ar_gid[6];      /* file member group identification */
        char    ar_mode[8];     /* file member mode */
        char    ar_size[10];    /* file member size */
        char    ar_fmag[2];     /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for *ar_mode* which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The *ar_name* filed is blank-padded and slash (/) terminated. The *ar_date* field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command *ar*(1) is used.

Each archive file member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero length name (i.e., **ar_name**[0] = '/'). The contents of this file are as follows:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: 4 bytes * "the number of symbols".
- The name string table. Length: *ar_size* – (4 bytes * ("the number of symbols" + 1)). The number of symbols and the array of offsets are managed with *sgetl* and *sputl*. The string table contains exactly as many null terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from

the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

ar(1), ld(1), strip(1), sputl(3X), a.out(4).

WARNINGS

Strip(1) will remove all archive symbol entries from the header. The archive symbol entries must be restored via the **s** option of the *ar*(1) command before the archive can be used with the link editor *ld*(1).

NAME

checklist – list of file systems processed by fsck

DESCRIPTION

Checklist resides in directory **/etc** and contains a list of at most 15 *special file*names. Each *special file*name is contained on a separate line and corresponds to a file system. If no *file-system* argument is provided to *fsck*(1M), each file listed in **/etc/checklist** is automatically read and checked for inconsistencies.

SEE ALSO

fsck(1M).

NAME

core – format of core image file

DESCRIPTION

The system writes out a core image of a terminated process when any of various errors occur. *Signal*(2) describes reasons for errors. The most common errors are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the working directory of the process (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The locations of the registers are outlined in **/usr/include/sys/reg.h**.

SEE ALSO

crash(1M), sdb(1), setuid(2), signal(2).

NAME

cpio – format of cpio archive

DESCRIPTION

When the **–c** option of *cpio*(1) is not used, the file header structure is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **–c** option is used, the header information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated pathname *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2).

NAME

/etc/cshrc-csh, ~/.cshrc – setting up a environment at C-shell startup time

DESCRIPTION

If a user's login directory contains a file named **.cshrc**, that file will be executed (via the C-shell's **source ~/.cshrc** command) before the shell starts accepting commands; **.cshrcs** are handy for setting aliases and C-shell variables. If the file **/etc/cshrc-csh** exists, it will be executed automatically for every user that does not have a **.cshrc**. For those who do have a **.cshrc**, it may be useful to add **source /etc/cshrc-csh** as the first line.

FILES

~/.cshrc
/etc/cshrc-csh

SEE ALSO

csh(1), login-csh(4).

NAME

dir – format of directories

SYNOPSIS

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its inode entry (see *fs*(4)). The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ      14
#endif
struct   direct
{
        ino_t  d_ino;
        char   d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for . and .. The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; because there is no parent, .. has the same meaning as ..

SEE ALSO

fs(4).

NAME

dump, ddate – incremental dump format

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>
#include <dumprestor.h>

DESCRIPTION

Tapes used by *dump*(1M) and *restor*(1M) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <*dumprestor.h*> is:

```
#define NTREC     10
#define MLEN      16
#define MSIZ      4096

#define TS_TAPE   1
#define TS_INODE  2
#define TS_BITS   3
#define TS_ADDR   4
#define TS_END    5
#define TS_CLRI   6
#define OFS_MAGIC   (int)60011
#define NFS_MAGIC   (int)60012
#define CHECKSUM(int)84446

#ifndef RESTORE
struct  spcl
{
        int         c_type;
        time_t      c_date;
        time_t      c_ddate;
        int         c_volume;
        daddr_t     c_tapea;
        ino_t       c_inumber;
        int         c_magic;
        int         c_checksum;
        struct      dinodec_dinode;
        int         c_count;
        char        c_addr[BSIZE];
} spcl;

struct  idates
{
        char        id_name[16];
        char        id_incno;
        time_t      id_ddate;
};
```

```
#else
/*
 * TP_BSIZE is the size of file blocks on the dump tapes.
 * Note that TP_BSIZE must be a multiple of DEV_BSIZE.
 *
 * NTREC is the number of TP_BSIZE blocks that are written
 * in each tape record.
 *
 * TP_NINDIR is the number of indirect pointers in a TS_INODE
 * or TS_ADDR record. Note that it must be a power of two.
 */
#define TP_BSIZE  1024
#define TP_NINDIR(TP_BSIZE/2)

union u_spcl {
        char dummy[TP_BSIZE];
        struct      s_spcl {
                    intc_type;
                    time_tc_date;
                    time_tc_ddate;
                    intc_volume;
                    daddr_tc_tapea;
                    ino_tc_inumber;
                    intc_magic;
                    intc_checksum;
                    structdinodec_dinode;
                    intc_count;
                    charc_addr[TP_NINDIR];
        } s_spcl;
} u_spcl;

#define spcl u_spcl.s_spcl
#endif

#defineDUMPOUTFMT"%-16s %c %s"/* for printf */
                    /* name, incno, ctime(date) */
#defineDUMPINFMT"%16s %c %[^00/* inverse for scanf */
```

NTREC is the number of 512 byte records in a physical tape block. *MLEN* is the number of bits in a bit map word. *MSIZ* is the number of bit map words.

The *TS_* entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR	A subrecord of a file description. See *c_addr* below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in *c_magic*.

CHECKSUM
Header records checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.
c_date The date the dump was taken.
c_ddate The date the file system was dumped from.
c_volume The current volume number of the dump.
c_tapea The current number of this (512-byte) record.
c_inumber The number of the inode being dumped if this is of type *TS_INODE*.
c_magic This contains the value *MAGIC* above, truncated as needed.
c_checksum
This contains whatever value is needed to make the record sum to *CHECKSUM*.
c_dinode This is a copy of the inode as it appears on the file system; see *fs*(4).
c_count The count of characters in *c_addr*.
c_addr An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, *TS_ADDR* records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a *TS_END* record and then the tapemark.

The structure *idates* describes an entry of the file */etc/ddate* where dump history is kept. The fields of the structure are:

id_name The dumped filesystem is '/dev/*id_nam*'.
id_incno The level number of the dump tape; see *dump*(1M).
id_ddate The date of the incremental dump in system format see *types*(5).

FILES

/etc/ddate

SEE ALSO

dump(1), dumpdir(1M), restor(1M), fs(4), types(5)

NAME

errfile – error-log file format

DESCRIPTION

When hardware errors are detected by the system, an error record is generated and passed to the error-logging daemon for recording in the error log for later analysis. The default error log is **/usr/adm/errfile**.

The format of an error record depends on the type of error that was encountered. Every record, however, has a header with the following format:

```
struct errhdr {
        short       e_type;     /* record type */
        short       e_len;      /* bytes in record (inc hdr) */
        time_t      e_time;     /* time of day */
};
```

The permissible record types are as follows:

```
#define E_GOTS          010     /* start for the UNIX/TS*/
#define E_GORT          011     /* start for the UNIX/RT*/
#define E_STOP          012     /* stop */
#define E_TCHG          013     /* time change */
#define E_CCHG          014     /* configuration change */
#define E_BLK           020     /* block device error */
#define E_STRAY         030     /* stray interrupt */
#define E_PRTY          031     /* memory parity */
```

Some records in the error file are of an administrative nature. These include the startup record that is entered into the file when logging is activated, the stop record that is written if the daemon is terminated "gracefully", and the time-change record that is used to account for changes in the system's time-of-day. These records have the following formats:

```
struct estart {
        short               e_cpu;      /* CPU type */
        struct utsname      e_name;     /* system names */
};

#define eend errhdr                     /* record header */

struct etimchg {
        time_t              e_ntime;    /* new time */
};
```

Stray interrupts cause a record with the following format to be logged:

```
struct estray {
        uint            e_saddr;        /* stray loc or device addr */
};
```

Generation of memory subsystem errors is not supported in this release.

Error records for block devices have the following format:

```
struct eblock {
```

```
        dev_t           e_dev;      /* "true" major + minor dev no */
        physadr         e_regloc;   /* controller address */
        short           e_bacty;    /* other block I/O activity */
        struct iostat {
            long        io_ops;     /* number read/writes */
            long        io_misc;    /* number "other" operations */
            ushort      io_unlog;   /* number unlogged errors */
        }               e_stats;
        short           e_bflags;   /* read/write, error, etc */
        short           e_cyloff;   /* logical dev start cyl */
        daddr_t         e_bnum;     /* logical block number */
        ushort          e_bytes;    /* number bytes to transfer */
        paddr_t         e_memadd;   /* buffer memory address */
        ushort          e_rtry;     /* number retries */
        short           e_nreg;     /* number device registers */
};
```

The following values are used in the *e_bflags* word:

```
#define E_WRITE     0       /* write operation */
#define E_READ      1       /* read operation */
#define E_NOIO      02      /* no I/O pending */
#define E_PHYS      04      /* physical I/O */
#define E_FORMAT    010     /* Formatting Disk*/
#define E_ERROR     020     /* I/O failed */
```

SEE ALSO

errdemon(1M).

NAME

filehdr – file header for common object files

SYNOPSIS

#include <filehdr.h>

DESCRIPTION

Every common object file begins with a 20-byte header. The following C **struct** declaration is used:

```
struct filehdr
{
        unsigned short  f_magic ;   /* magic number */
        unsigned short  f_nscns ;   /* number of sections */
        long            f_timdat ;  /* time & date stamp */
        long            f_symptr ;  /* file ptr to symtab */
        long            f_nsyms ;   /* # symtab entries */
        unsigned short  f_opthdr ;  /* sizeof(opt hdr) */
        unsigned short  f_flags ;   /* flags */
} ;
```

F_symptr is the byte offset into the file at which the symbol table can be found. Its value can be used as the offset in *fseek*(3S) to position an I/O stream to the symbol table. See *aouthdr*(4) for the structure of the optional aout header. The valid magic number is:

```
#define MC68MAGIC 0520    /* magic number */
```

The value in *f_timdat* is obtained from the *time*(2) system call. Flag bits currently defined are:

```
#define F_RELFLG  00001   /* relocation entries stripped */
#define F_EXEC    00002   /* file is executable */
#define F_LNNO    00004   /* line numbers stripped */
#define F_LSYMS   00010   /* local symbols stripped */
#define F_MINMAL  00020   /* minimal object file */
#define F_UPDATE  00040   /* update file, ogen produced */
#define F_SWABD   00100   /* file is "pre-swabbed" */
#define F_AR16WR  00200   /* 16-bit DEC host */
#define F_AR32WR  00400   /* 32-bit DEC host */
#define F_AR32W   01000   /* non-DEC host */
#define F_PATCH   02000   /* "patch" list in opt hdr */
```

SEE ALSO

time(2), fseek(3S), a.out(4), aouthdr(4).

NAME

file system – format of system volume

SYNOPSIS

```
#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>
```

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *superblock*. The format of a superblock is:

```
/*
 * Structure of the superblock
 */
struct   filsys
{
        ushort    s_isize;          /* size in blocks of i-list */
        daddr_t   s_fsize;          /* size in blocks of entire volume */
        short     s_nfree;          /* number of addresses in s_free */
        daddr_t   s_free[NICFREE];  /* free block list */
        short     s_ninode;         /* number of inodes in s_inode */
        ino_t     s_inode[NICINOD]; /* free inode list */
        char      s_flock;          /* lock during free list manipulation */
        char      s_ilock;          /* lock during i-list manipulation */
        char      s_fmod;           /* superblock modified flag */
        char      s_ronly;          /* mounted read-only flag */
        time_t    s_time;           /* last superblock update */
        short     s_dinfo[4];       /* device information */
        daddr_t   s_tfree;          /* total free blocks*/
        ino_t     s_tinode;         /* total free inodes */
        char      s_fname[6];       /* file system name */
        char      s_fpack[6];       /* file system pack name */
        long      s_fill[13];       /* ADJUST size of filsys to 512 */
        long      s_magic;          /* magic number to indicate new file system */
        long      s_type;           /* type of new file system */
};

#define FsMAGIC 0xfd187e20        /* s_magic number */
#define Fs1b    1                 /* 512-byte block */
#define Fs2b    2                 /* 1024-byte block */
```

S_type indicates the file system type. Currently, two types of file systems are supported: the original 512-byte oriented and the new improved 1024-byte oriented. *S_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *FsMAGIC*, the type is assumed to be *Fs1b*, otherwise the *s_type* field is used. In the following description, a block is then determined by the type. For the original 512-byte oriented file system, a block is 512 bytes. For the 1024-byte oriented file system, a block is 1024 bytes or two sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

S_isize is the address of the first data block after the i-list; the i-list starts just after the superblock, namely in block 2; thus the i-list is *s_isize*– 2 blocks long. *S_fsize* is the first block not

potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], ..., *s_free*[*s_nfree*– 1], up to 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

S_ninode is the number of free i-numbers in the *s_inode* array. To allocate an inode: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *s_inode* array, then try again. To free an inode, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed inode into any table. This list of inodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

S_tinode is the total free inodes available in the file system.

S_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

S_ronly is a read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

S_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for inodes begins in block 2. Also, inodes are 64 bytes long. Inode 1 is reserved for future use. Inode 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each inode represents one file. For the format of an inode and its flags, see *inode*(4).

FILES

/usr/include/sys/filsys.h
/usr/include/sys/stat.h

SEE ALSO

fsck(1M), fsdb(1M), mkfs(1M), inode(4).

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX System with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX System commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets <: and :>. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a – followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a – followed by the name of a "canned" tab specification.

Standard tabs are specified by **t– 8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t– 8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

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

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

SEE ALSO

ed(1), newform(1), tabs(1).

NAME

fstab – static information about the filesystems

SYNOPSIS

#include <fstab.h>

DESCRIPTION

The file */etc/fstab* contains descriptive information about the various file systems. */etc/fstab* is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. The order of records in */etc/fstab* is important because *fsck*, *mount*, and *umount* sequentially iterate through */etc/fstab* doing their thing.

The special file name is the **block** special file name, and not the character special file name. If a program needs the character special file name, the program must create it by appending a "r" after the last "/" in the special file name.

If *fs_type* is "rw" or "ro" then the file system whose name is given in the *fs_file* field is normally mounted read-write or read-only on the specified special file. If *fs_type* is "rq", then the file system is normally mounted read-write with disk quotas enabled. The *fs_freq* field is used for these file systems by the *dump*(8) command to determine which file systems need to be dumped. The *fs_passno* field is used by the *fsck*(8) program to determine the order in which file system checks are done at reboot time. The root file system should be specified with a *fs_passno* of 1, and other file systems should have larger numbers. File systems within a drive should have distinct numbers, but file systems on different drives can be checked on the same pass to utilize parallelism available in the hardware.

If *fs_type* is "sw" then the special file is made available as a piece of swap space by the *swapon*(8) command at the end of the system reboot procedure. The fields other than *fs_spec* and *fs_type* are not used in this case.

If *fs_type* is "rq" then at boot time the file system is automatically processed by the *quotacheck*(8) command and disk quotas are then enabled with *quotaon*(8). File system quotas are maintained in a file "quotas", which is located at the root of the associated file system.

If *fs_type* is specified as "xx" the entry is ignored. This is useful to show disk partitions which are currently not used.

```
#define FSTAB_RW    "rw"    /* read-write device */
#define FSTAB_RO    "ro"    /* read-only device */
#define FSTAB_RQ    "rq"    /* read-write with quotas */
#define FSTAB_SW    "sw"    /* swap device */
#define FSTAB_XX    "xx"    /* ignore totally */

struct fstab {
        char  *fs_spec;  /* block special device name */
        char  *fs_file;  /* file system path prefix */
        char  *fs_type;  /* rw,ro,sw or xx */
        int   fs_freq;   /* dump frequency, in days */
        int   fs_passno; /* pass number on parallel dump */
};
```

The proper way to read records from */etc/fstab* is to use the routines getfsent(), getfsspec(), getfstype(), and getfsfile().

FILES

/etc/fstab

SEE ALSO

getfsent(3X)

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. Lines that begin with # are ignored and may be used to comment the file. The format fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as *\nnn*, where *nnn* is the octal value of the desired character. The fields are:

label This is the string against which *getty*(1M) tries to match its second argument. It is often the speed at which the terminal is supposed to run, e.g., 1200, but it needn't be. If *getty*(1M) is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. The first entry is also used if *getty*(1M) can't find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

initial-flags These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*(1M). *Getty*(1M) understands the symbolic names specified in **/usr/include/sys/termio.h** (see *termio*(7). Normally only the speed flag is required in the *initial-flags* field. *Getty*(1M) automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty*(1M) executes *login*(1).

final-flags These flags take the same values as the *initial-flags* and are set just before *getty*(1M) executes *login*(1). The speed flag is again required. The composite flag SANE takes care of most of the other flags that need to be set so that the processor and terminal communicate in a rational fashion. The other two commonly specified *final-flags* are TAB3 (tabs are sent to the terminal as spaces) and HUPCL (the line is hung up on the final close).

login-prompt This entire field is printed as the *login-prompt*. White-space characters (space, tab, and new-line) are included in this field, unlike the other fields in which white space is ignored.

next-label This field indicates the next entry *label* in the table that *getty*(1M) should use if the user types a <*break*> or the input cannot be read. Usually, a series of speeds are linked together in a closed set. No matter where the set is entered, the correct speed can be obtained. For example, **2400** is linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

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

FILES

/etc/gettydefs

SEE ALSO

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

NAME

group – group file

DESCRIPTION

Group contains the following information for each group:

group name
encrypted password
numerical group ID
comma-separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group IDs to names.

FILES

/etc/group

SEE ALSO

newgrp(1), passwd(1), crypt(3C), passwd(4).

NAME

hostbin - binary host table

SYNOPSIS

#include <host.h>

DESCRIPTION

The *hostbin* file is composed of four sections:

An hbin structure indicating the offset and size of the hash tables.
For every host or network, an hdisk structure and trailing strings.
For every name, an hname structure, ordered and linked into a compressed hash table.
For every address, an haddr structure, ordered and linked into a compressed hash table.

The hbin structure looks like:

```
struct hbin
{
        int     hb_nnames;      /* Number of names */
        int     hb_naddrs;      /* Number of addresses */
        long    hb_hashoff;     /* Absolute file offset of name hash table */
};
```

Each host or network is represented in the binary file by the following structure:

```
struct hdisk
{
        short hd_capa;      /* Capability bits */
        short hd_naddrs;    /* Number of addresses */
        short hd_nnames;    /* Number of names */
        short hd_nprots;    /* Number of protocols */
};
```

followed by:

naddrs longword addresses
nnames null terminated name strings
a null terminated operating system string
nprotos null terminated protocol strings
a null terminated hardware string

The name and address hash table entries have the following structure:

```
struct hname
{
        char    hn_name[24];    /* Host or network name */
        long    hn_offset;      /* Absolute offset of host in file */
        long    hn_nhash;       /* Absolute offset of next hash entry */
};

struct haddr
{
        long    ha_addr;        /* Host or network address */
        long    ha_offset;      /* Absolute offset of host in file */
        long    ha_nhash;       /* Absolute offset of next hash entry */
};
```

The hash function *host_hash* in the host library is used to locate the starting offset of a hash chain in the name and address hash tables.

For the name table, this is

hbin.hb_hashoff + host_hash(name, strlen(name), hbin.hb_nnames) * sizeof (struct hname)

For the address table, this is

hbin.hb_hashoff + host_hash(& addr, sizeof addr, hbin.hb_naddrs) * sizeof (struct haddr)
+ hbin.hb_nnames * sizeof (struct hname)

A host is then found by comparing the desired name or address against hn_name or ha_addr, respectively. If unsuccessful, the next check is done by reading in the hname or haddr structure found at *hn_nhash* or *ha_nhash*. If this offset is zero, then the end of the chain has been reached, and no such name or address exists. For a few thousand names, a successful search match typically occurs on the second or third comparison.

REFERENCE

The Art of Computer Programming, Knuth, Volume 3, first edition, page 514 describes the algorithm used to generate the compressed hash tables.

FILES

/usr/include/host.h
/etc/hostbin

SEE ALSO

newhosts(1), host(3), myhostname(4)

NAME

inittab – script for the init process

DESCRIPTION

The **/etc/inittab** file supplies the script for *init*(1M) to perform as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process */etc/getty*, which initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

NOTE: Within this section, the term *init* always refers to the program described in *init*(1M).

The *inittab* file is composed of entries that are position-dependent and have the following format:

id:rstate:action:process

Each entry is delimited by a new-line; however, a backslash (\) preceding a new-line indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *gettys* are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This field is 1 to 4 characters used to uniquely identify an entry.

rstate This field defines the *run-level* in which this entry is to be processed. *Run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field will be processed. When *init* is requested to change *run-levels*, all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0**– **6**. If no *run-level* is specified, *action* will be taken on this *process* for all *run-levels*, **0**– **6**. There are three other values, **a**, **b**, and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* (see *init*(1M)) process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that the system is only in these states for as long as it takes to execute all the entries associated with the states. A process started by an **a**, **b**, or **c** command is not killed when *init* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn If the process does not exist, *init* is to start the process, not wait for its termination (continue scanning the *inittab* file), and, when it dies, restart the process. If the process currently exists *init* is to do nothing and continue scanning the *inittab* file.

wait When *init* enters the *run-level* that matches the entry's *rstate*, it is to start the process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once When *init* enters a *run-level* that matches the entry's *rstate*, it is to start the process, not wait for its termination and, when it dies, not restart the process. If a new *run-level* is entered when the process is still running, the program will not be restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, not wait for its termination, and, when it dies, not restart the process. In order for this instruction to be meaningful, either the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *Init* is to start the process, wait for its termination, and, when it dies, not restart the process.

powerfail *Init* is to execute the process associated with this entry only when it receives a powerfail signal (SIGPWR; see *signal*(2)).

powerwait *Init* is to execute the process associated with this entry only when it receives a powerfail signal (SIGPWR) and is to wait until the process terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, *init* is to send the warning signal (SIGTERM) and wait 20 seconds before forcibly terminating the process via the kill signal (SIGKILL). If the process is nonexistent, *init* is to ignore the entry.

ondemand This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b**, or **c** values described in the *rstate* field.

initdefault An entry with this *action* is scanned only when *init* is initially invoked. *Init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and *init* will enter *run-level* **6**. If the **initdefault** entry is **s**, *init* will start in the *SINGLE USER* state. If *init* doesn't find an **initdefault** entry in **/etc/inittab**, it will request an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console. It is expected that this entry will be only used to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh – c** *'exec command'*. For this reason, any legal *sh* syntax can appear in the the *process* field. Comments can be inserted with the ; *#comment* syntax.

FILES

/etc/inittab

SEE ALSO

getty(1M), init(1M), sh(1), who(1), exec(2), open(2), signal(2).

NAME

inode – format of an inode

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>

DESCRIPTION

An inode for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>**.

```
/* Inode structure as it appears on a disk block. */
struct dinode
{
        ushort  di_mode;    /* mode and type of file */
        short   di_nlink;   /* number of links to file */
        ushort  di_uid;     /* owner's user id */
        ushort  di_gid;     /* owner's group id */
        off_t   di_size;    /* number of bytes in file */
        char    di_addr[40]; /* disk block addresses */
        time_t  di_atime;   /* time last accessed */
        time_t  di_mtime;   /* time last modified */
        time_t  di_ctime;   /* time created */
};
/*
 * the 40 address bytes:
 *     39 used; 13 addresses
 *     of 3 bytes each.
 */
```

For the meaning of the defined types *off_t* and *time_t*, see *types*(5).

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(4), types(5).

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by *getty*(1M) and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

getty(1M), login(1).

NAME

ldfcn – common object file access routines

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

DESCRIPTION

The common object file access routines are a collection of functions for reading an object file that is in common object file form. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type LDFILE (defined as **struct ldfile**), which is declared in the header file **<ldfcn.h>**. The primary purpose of this structure is to provide uniform access to both simple object files and object files that are members of an archive file.

The function *ldopen*(3X) allocates and initializes the LDFILE structure and returns a pointer to the structure to the calling program. The fields of the LDFILE structure may be accessed individually through macros defined in **<ldfcn.h>** and contain the following information:

LDFILE *ldptr;

TYPE(ldptr) The file magic number, used to distinguish between archive members and simple object files.

IOPTR(ldptr) The file pointer returned by *fopen*(3S) and used by the standard input/output functions.

OFFSET(ldptr) The file address of the beginning of the object file; the offset is non-zero if the object file is a member of an archive file.

HEADER(ldptr) The file header structure of the object file.

The object file access functions may be divided into four categories:

(1) functions that open or close an object file

ldopen(3X) and *ldaopen*
open a common object file
ldclose(3X) and *ldaclose*
close a common object file

(2) functions that read header or symbol table information

ldahread(3X)
read the archive header of a member of an archive file
ldfhread(3X)
read the file header of a common object file
ldshread(3X) and *ldnshread*
read a section header of a common object file
ldtbread(3X)
read a symbol table entry of a common object file
ldgetname(3X)
retrieve a symbol name from a symbol table entry or from the string table

(3) functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section.

ldohseek(3X)
seek to the optional file header of a common object file
ldsseek(3X) and *ldnsseek*
seek to a section of a common object file
ldrseek(3X) and *ldnrseek*
seek to the relocation information for a section of a common object file
ldlseek(3X) and *ldnlseek*
seek to the line number information for a section of a common object file
ldtbseek(3X)
seek to the symbol table of a common object file

(4) the function *ldtbindex*(3X) which returns the index of a particular common object file symbol table entry

These functions are described in detail in the manual pages identified for each function.

All the functions except *ldopen*, *ldaopen*, and *ldtbindex* return either SUCCESS or FAILURE, which are constants defined in **<ldfcn.h>**. *Ldopen* and *ldaopen* both return pointers to a LDFILE structure.

MACROS

Additional access to an object file is provided through a set of macros defined in **<ldfcn.h>**. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the LDFILE structure into a reference to its file descriptor field.

The following macros are provided:

GETC(ldptr)
FGETC(ldptr)
GETW(ldptr)
UNGETC(c, ldptr)
FGETS(s, n, ldptr)
FREAD((char *) ptr, sizeof (*ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)
STROFFSET(ldptr)

The STROFFSET macro calculates the address of the string table in a object file. See the manual entries for the corresponding standard input/output library functions for details on the use of these macros. (The functions are identified as 3S in Section 3 of this manual.)

The program must be loaded with the object file access routine library **libld.a**.

WARNINGS

The macro FSEEK defined in the header file **<ldfcn.h>** translates into a call to the standard input/output function *fseek*(3S). FSEEK should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members.

SEE ALSO

fopen(3S), fseek(3S), ldahread(3X), ldclose(3X), ldfhread(3X), ldgetname(3X), ldlread(3X), ldlseek(3X), ldohseek(3X), ldopen(3X), ldrseek(3X), ldlseek(3X), ldshread(3X), ldtbindex(3X), ldtbread(3X), ldtbseek(3X).

Common Object File Format, by I. S. Law.

NAME

linenum – line number entries in a common object file

SYNOPSIS

#include <linenum.h>

DESCRIPTION

The C compiler generates an entry in the object file for each C source line on which a breakpoint is possible (when invoked with the **–g** option; see *cc*(1)). Users can then reference line numbers when using the appropriate software test system (see *sdb*(1)). The structure of these line number entries appears below.

```
struct lineno
{
        union
        {
                long            l_symndx ;
                long            l_paddr ;
        }                       l_addr ;
        unsigned short          l_lnno ;
};
```

Numbering starts with one for each function. The initial line number entry for a function has *l_lnno* equal to zero, and the symbol table index of the function's entry is in *l_symndx*. Otherwise, *l_lnno* is non-zero, and *l_paddr* is the physical address of the code for the referenced line. Thus the overall structure is the following:

l_addr	*l_lnno*
function symtab index	0
physical address	line
physical address	line
...	
function symtab index	0
physical address	line
physical address	line
...	

SEE ALSO

cc(1), sdb(1), a.out(4).

NAME

master – master device information table

DESCRIPTION

This file is used by the *config*(1M) program to obtain device information that enables it to generate the configuration files. The file consists of 3 or 4 parts, each separated by a line with a dollar sign ($) in column 1. Part 1 contains device information; part 2 contains names of devices that have aliases; part 3 contains tunable parameter information. Part 4 is optional and contains information related to configuring the M68000 family systems only. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1 contains lines consisting of at least 10 fields and at most 13 fields. The fields are delimited by tabs and/or blanks.

Field 1: device name (8 chars. maximum).

Field 2: interrupt vector size (decimal, in bytes).

Field 3: device mask (octal)– each "on" bit indicates that the handler exists:

004000 block device print routine
002000 block device strategy routine
001000 block device close routine
000400 block device open handler
000200 tty header exists
000100 initialization handler
000040 power-failure handler
000020 character device open handler
000010 character device close handler
000004 character device read handler
000002 character device write handler
000001 character device ioctl handler

Field 4: device type indicator (octal):

002000 suppress argument passing to interrupt handler routine
001000 suppress interrupt subroutine call
000400 suppress interrupt routine, but allow interrupt vector
000200 allow only one of these devices
000100 suppress count field in the **conf.c** file
000040 suppress interrupt vector
000020 required device
000010 block device
000004 character device
000002 interrupt driven device other than block or char. device

Field 5: handler prefix (4 chars. maximum).

Field 6: device address size (decimal).

Field 7: major device number for block-type device.

Field 8: major device number for character-type device.

Field 9: maximum number of devices per controller (decimal).

Field 10: maximum bus request level (1 through 7).

Fields 11-13: optional configuration table structure declarations (8 chars. maximum)

Part 2 contains lines with 2 fields each:

Field 1: alias name of device (8 chars. maximum).

Field 2: reference name of device (8 chars. maximum; specified in part 1).

Part 3 contains lines with 2 or 3 fields each:

Field 1: parameter name (as it appears in description file; 30 chars. maximum)

Field 2: parameter name (as it appears in the **conf.c** file; 30 chars. maximum)

Field 3: default parameter value (30 chars. maximum; parameter specification is required if this field is omitted)

Part 4 contains M68000-specific lines exactly like those for the M68000-specific portion of the **dfile**. See *config* (1M) for a description of these lines.

Devices that are not interrupt-driven have an interrupt vector size of zero. The 040 bit in Field 4 causes *config*(1M) to record the interrupt vectors although the **locore.s** file will show no interrupt vector assignment at those locations (interrupts here will be treated as strays).

SEE ALSO

config(1M).

NAME

mnttab – mounted file system table

SYNOPSIS

#include <mnttab.h>

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices, mounted by the *mount*(1M) command, in the following structure as defined by **<mnttab.h>**:

```
struct  mnttab {
        char        mt_dev[10];
        char        mt_filsys[10];
        short       mt_ro_flg;
        time_t      mt_time;
};
```

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the *special file* is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file*'s read/write permissions and the date on which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter NMOUNT located in **/usr/src/uts/cf/conf.c**, which defines the number of allowable mounted special files.

SEE ALSO

mount(1M), setmnt(1M).

NAME

mtab – mounted file system table

SYNOPSIS

#include <fstab.h>
#include <mtab.h>

DESCRIPTION

Mtab resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

The table is a series of *mtab* structures, as defined in <mtab.h>. Each entry contains the null-padded name of the place where the special file is mounted, the null-padded name of the special file, and a type field, one of those defined in *<fstab.h>*. The special file has all its directories stripped away; that is, everything through the last '/' is thrown away. The type field indicates if the file system is mounted read-only, read-write, or read-write with disk quotas enabled.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(1M)

NAME

myhostname – Specification of this host's name

DESCRIPTION

The file */etc/myhostname* contains a single linefeed-terminated line with this host's network name. The name may be the primary host name, or any of its nicknames. Only one name need be specified; the rest are found by the host(3) library by looking in the binary host table hostbin(5). The case of the name is unimportant.

/etc/myhostname is read directly by the host(3) library, and is the reason binary host tables may be copied between different systems of the same "Indian-ness" (byte ordering). Note, however, that the name specified in /etc/myhostname must be one of those in the compiled host table. Thus, /etc/myhostname is used to select which host in the table is "me", not to add local information to a generic table.

FILES

/etc/myhostname

SEE ALSO

newhosts(1), host(3), hostbin(4)

NAME

passwd – password file

DESCRIPTION

Passwd contains the following information for each user:

login name
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user entry is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user IDs to names.

The encrypted password consists of 13 characters chosen from a 64-character alphabet (., /, **0**– **9**, **A**– **Z**, **a**– **z**). If the password is null, the encrypted password is also null. Password aging is effected for a particular user if the encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. Such a string must be introduced in the first instance by the superuser.

The first character of the password age, e.g., *M*, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after the password has expired will be forced to supply a new one. The next character, e.g., *m*, denotes the minimum period (in weeks) which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. A null string is equivalent to zero. *M* and *m* have numerical values in the range 0– 63 that correspond to the 64-character alphabet shown above (i.e., / = 1 week; **z** = 63 weeks). If *m* = *M* = 0 (derived from the string . or ..) the password must be changed the next time the user logs in (and the "age" will disappear from the user's entry in the password file). If *m* > *M* (signified, e.g., by the string ./), only the superuser will be able to change the password.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a64l(3C), crypt(3C), getpwent(3C), group(4).

NAME

phones – remote host phone number data base

DESCRIPTION

The file /etc/phones contains the system-wide private phone numbers for the *tip*(1C) program. This file is normally unreadable, and so may contain privileged information. The format of the file is a series of lines of the form: <system-name>[\t]*<phone-number>. The system name is one of those defined in the *remote*(4) file and the phone number is constructed from [0123456789-=*%]. The "=" and "*" characters are indicators to the auto call units to pause and wait for a second dial tone (when going through an exchange). The "=" is required by the DF02-AC and the "*" is required by the BIZCOMP 1030.

Only one phone number per line is permitted. However, if more than one line in the file contains the same system name *tip*(1C) will attempt to dial each one in turn, until it establishes a connection.

FILES

/etc/phones

SEE ALSO

tip(1C), remote(4)

NAME

pnch – file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

SEE ALSO

send(1C).

NAME

profile – setting up an environment at login time

DESCRIPTION

If a user's login directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before the user's session begins; **.profiles** are handy for setting exported environment variables and terminal modes. If the file **/etc/profile** exists, it will be executed for every user before the **.profile**. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
echo "terminal: \c"
read TERM
case $TERM in
        300)        stty cr2 nl0 tabs; tabs;;
        300s)       stty cr2 nl0 tabs; tabs;;
        450)        stty cr2 nl0 tabs; tabs;;
        hp)         stty cr0 nl0 tabs; tabs;;
        745|735)    stty cr1 nl1 - tabs; TERM=745;;
        43)         stty cr1 nl0 - tabs;;
        4014|tek)   stty cr0 nl0 - tabs ff1; TERM=4014; echo "\33;";;
        *)          echo "$TERM unknown";;
esac
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(5), term(5).

NAME

reloc – relocation information for a common object file

SYNOPSIS

#include <reloc.h>

DESCRIPTION

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format.

```
struct    reloc
{
        long        r_vaddr ;    /* (virtual) address of reference */
        long        r_symndx ;   /* index into symbol table */
        short       r_type ;     /* relocation type */
};


/*
 * All generics
 *      reloc. already performed to symbol in the same section
 */
#define  R_ABS                  0

/*
 * DEC Processors  VAX 11/780 and VAX 11/750
 *
 */
#define R_RELBYTE               017
#define R_RELWORD               020
#define R_RELLONG               021
#define R_PCRBYTE               022
#define R_PCRWORD               023
#define R_PCRLONG               024

/*
 * Motorola 68000 uses R_RELBYTE, R_RELWORD, R_RELLONG,
 * R_PCRBYTE, and R_PCRWORD as for DEC machines above.
 */
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

R_ABS	The reference is absolute, and no relocation is necessary. The entry will be ignored.
R_RELBYTE	A direct 8-bit reference to a symbol's virtual address.
R_RELWORD	A direct 16-bit reference to a symbol's virtual address.
R_RELLONG	A direct 32-bit reference to a symbol's virtual address.
R_PCRBYTE	A "PC-relative" 8-bit reference to a symbol's virtual address.
R_PCRWORD	A "PC-relative" 16-bit reference to a symbol's virtual address.
R_PCRLONG	A "PC-relative" 32-bit reference to a symbol's virtual address.

On the VAX processors, relocation of a symbol index of -1 indicates that the relative difference between the current segment's start address and the program's load address is added to the relocatable address.

Other relocation types will be defined as they are needed.

Relocation entries are generated automatically by the assembler and automatically utilized by the link editor. A link editor option exists for removing the relocation entries from an object file.

SEE ALSO

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

NAME

remote – remote host description file

DESCRIPTION

The systems known by *tip*(1C) and their attributes are stored in an ASCII file which is structured somewhat like the *termcap*(4) file. Each line in the file provides a description for a single *system*. Fields are separated by a colon (":"). Lines ending in a \ character with an immediately following newline are continued on the next line.

The first entry is the name(s) of the host system. If there is more than one name for a system, the names are separated by vertical bars. After the name of the system comes the fields of the description. A field name followed by an '=' sign indicates a string value follows. A field name followed by a '#' sign indicates a following numeric value.

Entries named "tip*" and "cu*" are used as default entries by *tip*, and the *cu* interface to *tip*, as follows. When *tip* is invoked with only a phone number, it looks for an entry of the form "tip300", where 300 is the baud rate with which the connection is to be made. When the *cu* interface is used, entries of the form "cu300" are used.

CAPABILITIES

Capabilities are either strings (str), numbers (num), or boolean flags (bool). A string capability is specified by *capability=value*; e.g. "dv=/dev/harris". A numeric capability is specified by *capability#value*; e.g. "xa#99". A boolean capability is specified by simply listing the capability.

at (str) Auto call unit type.

br (num) The baud rate used in establishing a connection to the remote host. This is a decimal number. The default baud rate is 300 baud.

cm (str) An initial connection message to be sent to the remote host. For example, if a host is reached through port selector, this might be set to the appropriate sequence required to switch to the host.

cu (str) Call unit if making a phone call. Default is the same as the 'dv' field.

di (str) Disconnect message sent to the host when a disconnect is requested by the user.

du (bool) This host is on a dial-up line.

dv (str) UNIX device(s) to open to establish a connection. If this file refers to a terminal line, *tip*(1C) attempts to perform an exclusive open on the device to insure only one user at a time has access to the port.

el (str) Characters marking an end-of-line. The default is NULL. '~' escapes are only recognized by *tip* after one of the characters in 'el', or after a carriage-return.

fs (str) Frame size for transfers. The default frame size is equal to BUFSIZ.

hd (bool) The host uses half-duplex communication; local echo should be performed.

ie (str) Input end-of-file marks. The default is NULL.

oe (str) Output end-of-file string. The default is NULL. When *tip* is transferring a file, this string is sent at end-of-file.

pa (str) The type of parity to use when sending data to the host. This may be one of "even", "odd", "none", "zero" (always set bit 8 to zero), "one" (always set bit 8 to 1). The default is even parity.

pn (str) Telephone number(s) for this host. If the telephone number field contains an @ sign, *tip* searches the file */etc/phones* file for a list of telephone numbers; c.f. *phones*(4).

tc (str) Indicates that the list of capabilities is continued in the named description. This is used primarily to share common capability information.

Here is a short example showing the use of the capability continuation feature:

```
UNIX-1200:\
        :dv=/dev/cau0:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:br#1200:
arpavax|ax:\
        :pn=7654321%:tc=UNIX-1200
```

FILES

/etc/remote

SEE ALSO

tip(1C), phones(4)

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of 6 logical parts: the *checksum*, the *delta table* (information about each delta), *user names* (login names and/or numerical group IDs of users who may add deltas), *flags* (definitions of internal keywords), *comments* (arbitrary descriptive information about the file), and the *body* (the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a 5-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@ hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@ h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

@ s DDDDD/DDDDD/DDDDD
@ d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@ i DDDDD ...
@ x DDDDD ...
@ g DDDDD ...
@ m <MR number>
.
.
.
@ c <comments> ...
.
.
.
@ e

The first line (**@ s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@ d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor.

The **@ i**, **@ x**, and **@ g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@ m** lines (optional) each contain one MR number associated with the delta; the **@ c** lines contain comments associated with the delta.

The **@ e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @**u** and @**U**. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

@**f** <flag> <optional text>

The following flags are defined:

@**f** t <type of program>
@**f** v <program name>
@**f** i
@**f** b
@**f** m <module name>
@**f** f <floor>
@**f** c <ceiling>
@**f** d <default-sid>
@**f** n
@**f** j
@**f** l <lock-releases>
@**f** q <user defined>
@**f** z <reserved for use in interfaces>

The **t** flag defines the replacement for the %Y% identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present, it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "**No id keywords**" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (i.e., the file will not be gotten, or the delta will not be made). When the **b** flag is present, the **-b** keyletter may be used with the *get* command to create a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the %M% identification keyword. The **f** flag defines the "floor" release, i.e., the release below which no deltas may be added. The **c** flag defines the "ceiling" release, i.e., the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **-e** keyletter). The **q** flag defines the replacement for the %Q% identification keyword. The **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text surrounded by the bracketing lines @**t** and @**T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character; only control lines do. There are three kinds of control lines:

insert, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide in the *User's Guide*.

NAME

scnhdr – section header for a common object file

SYNOPSIS

#include <scnhdr.h>

DESCRIPTION

Every common object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The C structure appears below.

```
struct scnhdr
{
        char            s_name[SYMNMLEN]; /* section name */
        long            s_paddr;    /* physical address */
        long            s_vaddr;    /* virtual address */
        long            s_size;     /* section size */
        long            s_scnptr;   /* file ptr to raw data */
        long            s_relptr;   /* file ptr to relocation */
        long            s_lnnoptr;  /* file ptr to line numbers */
        unsigned short  s_nreloc;   /* # reloc entries */
        unsigned short  s_nlnno;    /* # line number entries */
        long            s_flags;    /* flags */
};
```

File pointers are byte offsets into the file; they can be used as the offset in a call to *fseek*(3S). If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it, but it can have no relocation entries, line numbers, or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr*, *s_relptr*, *s_lnnoptr*, *s_nreloc*, and *s_nlnno* are zero.

SEE ALSO

ld(1), fseek(3S), a.out(4).

NAME

syms – common object file symbol table format

SYNOPSIS

#include <syms.h>

DESCRIPTION

Common object files contain information to support *symbolic* software testing (see *sdb*(1). Line number entries, *linenum*(4), and extensive symbolic information permit testing at the C *source* level. Every object file's symbol table is organized as shown below.

```
Filename 1.
        Function 1.
                Local symbols for function 1.
        Function 2
                Local symbols for function 2.
        ...
        Static externs for file 1.

Filename 2.
        Function 1.
                Local symbols for function 1.
        Function 2.
                Local symbols for function 2.
        ...
        Static externs for file 2.
...

Defined global symbols.
Undefined global symbols.
```

The entry for a symbol is a fixed-length structure. The members of the structure hold the name (null padded), its value, and other information. The C structure is given below.

```
#define SYMNMLEN    8
#define FILNMLEN    14

struct    syment
{
  union                                 /* ways to get a symbol name*/
  {
        char            _n_name[SYMNMLEN] ;/* names less than 8 chars. */
        struct                          /* names 8 char or more*/
        {
          long          _n_zeroes;      /* === 0L when in string table*/
          long          _n_offset;      /* location of name in table */
        }n_n;
        char            *_n_nptr[2];    /* allows overlaying */
  }_n;
        long            n_value ;       /* value of symbol */
        short           n_scnum ;       /* section number */
        unsigned short  n_type ;        /* type and derived type */
        char            n_sclass ;      /* storage class */
        char            n_numaux ;      /* number of aux entries */
};
#define n_name          _n._n_name
```

```
#define n_zeroes        _n._n_n._n_zeroes
#define n_offset        _n._n_n._n_offset
#define n_nptr          _n._n_nptr[1]
```

Meaningful values and explanations for them are given in both **syms.h** and *Common Object File Format*. Anyone who needs to interpret the entries should seek more information in these sources. Some symbols require more information than a single entry; they are followed by *auxiliary entries* that are the same size as a symbol entry. The format follows.

```
union auxent
{
        struct
        {
                long            x_tagndx;
                union
                {
                        struct
                        {
                                unsigned short  x_lnno;
                                unsigned short  x_size;
                        } x_lnsz;
                        long    x_fsize;
                } x_misc;
                union
                {
                        struct
                        {
                                long .  x_lnnoptr;
                                long    x_endndx;
                        }       x_fcn;
                        struct
                        {
                                unsigned short  x_dimen[DIMNUM];
                        }       x_ary;
                }               x_fcnary;
                unsigned short  x_tvndx;
        }       x_sym;
        struct
        {
                char    x_fname[FILNMLEN];
        }       x_file;
        struct
        {
                long    x_scnlen;
                unsigned short  x_nreloc;
                unsigned short  x_nlinno;
        }       x_scn;

        struct
        {
                unsigned short  x_tvlen;
                unsigned short  x_tvran[2];
        }       x_tv;
};
```

Indexes of symbol table entries begin at *zero*.

SEE ALSO

sdb(1), a.out(4), linenum(4).
Common Object File Format by I. S. Law.

WARNING

Longs are equivalent to ints and are converted to ints in the compiler to minimize the complexity of the compiler code generator. Thus, the information about which symbols are declared as longs and which as ints cannot be determined from the symbol table.

NAME

tar – tape archive file format

DESCRIPTION

Tar, (the tape archive command) dumps several files into one, in a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the *tar*(1) command line — default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads.

The header block looks like:

```
#define TBLOCK      512
#define NAMSIZ      100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width w) contains w-2 digits, a space, and a null, except *size* and *mtime*, which do not contain the trailing null. *Name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the top bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped. *Chksum* is a decimal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is ASCII '0' if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '2' if it is a symbolic link. The name linked-to, if any, is in *linkname*, with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually rescanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1)

BUGS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

NAME

termcap – terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base which describes terminals. Each entry in the file gives a set of capabilities for a terminal and describes how operations are performed. Padding requirements and initialization sequences are included in *termcap*. The data base is used by programs such as *vi*(1).

Entries in *termcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older systems which store the terminal type in a 16-bit word in a systemwide data base. The second name is the most common abbreviation for the terminal and the last name should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

Preparing Descriptions

The most effective way to prepare a terminal description is to imitate the description of a similar terminal in *termcap* and build up a description gradually, using partial descriptions with *ex* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *termcap* file to describe it or bugs in *ex*. To easily test a new terminal description, set the environment variable TERMCAP to a pathname of a file containing the description being worked on; the editor will look there rather than in **/etc/termcap.** TERMCAP can also be set to the *termcap* entry itself to avoid reading the file when starting up the editor.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

hn |2621nl:ks@ :ke@ :tc=2621:

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

CAPABILITIES

Capabilities in *termcap* are of three types: Boolean capabilities, which indicate that the terminal has some particular feature; numeric capabilities, which give the size of the terminal or the size of particular delays; and string capabilities, which give a sequence that can be used to perform particular terminal operations.

Entries may be continued onto multiple lines by giving a \ as the last character of a line. Empty fields may be included for readability (e.g., between the last field on a line and the first field on the next).

List of Capabilities

(P) indicates padding may be specified
(P*) indicates that padding may be based on number of lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace character, if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command char in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horiz motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give :ei=: if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give :im=: if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by other function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of keypad transmit mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of other keys

ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in keypad transmit mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on other function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by *vi* version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though no overstrike
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
xb	bool		Beehive (f1=escape, f2=ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like **ce** \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept– 100, is among the more complex entries in the *termcap* file as of this writing. (This particular Concept entry is outdated, and is used as an example only.)

```
c1 |c100 |concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
   :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
   :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:nd=\E=:\
   :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Capability Descriptions

All capabilities have 2-letter codes. For instance, the fact that the Concept-100 has automatic margins (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am** in the sample description above. Numeric capabilities are followed by the character '#' and then the value. Thus, **co**, which indicates the number of columns the terminal has, gives the value '80' for the Concept-100.

String-valued capabilities, such as **ce** (clear to end of line sequence), are given by the 2-character code, an '=', and a string ending at the next field separator (:). A delay in milliseconds may appear after the '=' in such a capability and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either an integer, e.g., '20', or an integer followed by an '*', i.e., '3*'. An '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When an '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string-valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as 3 octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a colon (:) in a capability, it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability, it must be encoded as **\200**. The routines which deal with *termcap* use C strings, and strip the high bits of the output very late; therefore, a **\200** comes out as a **\000** would.

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, its description should include the **am** capability. If the terminal can clear its screen, this is given by the **cl** string capability. If the terminal can backspace, it should have the **bs** capability, unless a backspace is accomplished by a character other than ^H, in which case the alternate character should be given as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over), it should have the **os** capability.

A very important point is that the local cursor motions encoded in *termcap* are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch-selectable automatic margins, the *termcap* file usually assumes that this is on, i.e., **am**.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus, the model 33 teletype is described as

```
t3 |33 |tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl |adm3β|si adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by the **cm** string capability. It uses escapes like those in *printf*(3s), i.e., %x. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and column to which motion is desired. The

%encodings have the following meanings:

%d	as in *printf*, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds *x* to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

For example, to get to row 3 and column 12 the HP2645 needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as 2 digits. Thus, its **cm** capability is cm=6\E&%r%2c%2Y. The Microterm ACT-IV needs the current row and column sent, preceded by a ^T, with the row and column simply encoded in binary, cm=^T%.%.. Terminals which use %. need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** is introduced below). This is necessary because it is not always safe to transmit **\t**, **\n** ^**D** and **\r**, because the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character; thus, cm=\E=%+ %+ .

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to the very upper left corner of screen), this can be given as **ho**; similarly, a fast way of getting to the lower left hand corner can be given as **ll**; this may involve moving up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, this can be given as **sb**, although just **al** suffices. If the terminal can retain display memory above, the **da** capability should be given; if display memory can be retained below, **db** should be given. These capabilities let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

Termcap can be used to describe two basic kinds of intelligent terminals with respect to insert/delete characters. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin-Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen; the blank is either eliminated or expanded to 2 untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals with an insert mode that does not fall into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) To specify **im**, give the sequence to get into insert mode or give an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode. If you gave **im** with an empty value, give **ei** with an empty value also. Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**; terminals which send a sequence to open a screen position should give it here. If post-insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode; give **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining – half bright is not usually an acceptable standout mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves 1 or even 2 blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **ug** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **us** and **ue**, respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes, respectively. These can be used to change, e.g., from an underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This need arises, for example, from terminals like the Concept-100 with more than one page of memory. If the terminal has only memory-relative cursor addressing and not screen relative cursor addressing, a 1-screen sized window must be fixed into the terminal for cursor addressing to work properly.

If the terminal correctly generates underlined characters (with no special codes needed), even though it does not overstrike, you should give the capability **ul**. If overstrikes are erasable with a blank, this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad works only in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**; otherwise, the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh**, respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2-letter codes can be given in the **ko** capability. For example, :ko=cl,ll,sf,sb: says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the **cl, ll, sf**, and **sb** entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of *vi*, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh**. It consists of groups of 2 characters. In each group, the first character is what an arrow key sends, the second character is the corresponding *vi* command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the Mime would be **:ma=^Kj^Zk^Xl:**, indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the Mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed, should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed, should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x***x*.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

NOTE

Termcap is based on software developed by the University of California, Berkeley, California, Computer Science Division, Department of Electrical Engineering and Computer Science.

Termcap will be replaced by *terminfo* in the next release. Transition tools will be provided.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), vi(1), termcap(3)

WARNINGS AND BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap(3)* do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1,024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

#include <sys/types.h>
#include <utmp.h>

DESCRIPTION

These files hold user and accounting information for commands such as *who*(1), *write*(1), and *login*(1). They have the following structure, as defined by **<utmp.h>**:

```
#define   UTMP_FILE    "/etc/utmp"
#define   WTMP_FILE    "/etc/wtmp"
#define   ut_name      ut_user

struct utmp {
      char      ut_user[8];          /* User login name */
      char      ut_id[4];            /* /etc/inittab id (usually line #) */
      char      ut_line[12];         /* device name (console, lnxx) */
      short     ut_pid;              /* process id */
      short     ut_type;             /* type of entry */
      struct    exit_status {
         short     e_termination;   /* Process termination status */
         short     e_exit;          /* Process exit status */
      } ut_exit;                     /* The exit status of a process
                                      * marked as DEAD_PROCESS. */
      time_t    ut_time;             /* time entry was made */
};

/* Definitions for ut_type */
#define EMPTY          0
#define RUN_LVL        1
#define BOOT_TIME      2
#define OLD_TIME       3
#define NEW_TIME       4
#define INIT_PROCESS   5              /* Process spawned by "init" */
#define LOGIN_PROCESS  6              /* getty process waiting for login */
#define USER_PROCESS   7              /* A user process */
#define DEAD_PROCESS   8
#define ACCOUNTING     9
#define UTMAXTYPE      ACCOUNTING  /* Largest legal ut_type */

/* Special strings or formats used in the "ut_line" field when  */
/* accounting for something other than a process.  */
/* No string for the ut_line field can be more than 11 chars +  */
/* a NULL in length.  */
#define RUNLVL_MSG  "run- level %c"
#define BOOT_MSG    "system boot"
#define OTIME_MSG   "old time"
#define NTIME_MSG   "new time"
```

FILES

/usr/include/utmp.h
/etc/utmp
/etc/wtmp

SEE ALSO

login(1), who(1), write(1), getut(3C).

NAME

intro – introduction to miscellaneous facilities

DESCRIPTION

This section describes facilities such as formatting documentation and setting the terminal environment. It also contains descriptions of various character set tables, flag values, and user-accessible data types.

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
010 bs	011 ht	012 nl	013 vt	014 np	015 cr	016 so	017 si
020 dle	021 dc1	022 dc2	023 dc3	024 dc4	025 nak	026 syn	027 etb
030 can	031 em	032 sub	033 esc	034 fs	035 gs	036 rs	037 us
040 sp	041 !	042 "	043 #	044 $	045 %	046 &	047 '
050 (	051)	052 *	053 +	054 ,	055 -	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 del

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [	5c \	5d]	5e ^	5f _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c |	7d }	7e ~	7f del

FILES

/usr/pub/ascii

NAME

environ – user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form *name=value*. The following names are used by various commands:

PATH The sequence of directory prefixes that commands such as *sh*(1), *time*(1), *nice*(1), and *nohup*(1) apply in searching for a file known by an incomplete pathname. The prefixes are separated by colons (**:**). *Login*(1) sets **PATH=:/bin:/usr/bin**.

HOME Name of the user's login directory, set by *login*(1) from the password file *passwd*(4).

TERM The kind of terminal for which output is to be prepared. This information is used by commands such as *mm*(1), *vi*(1), and *tplot*(1G), which may exploit special capabilities of the terminal.

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is the standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

Further names may be placed in the environment by the *export* command and *name=value* arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files, e.g., **MAIL**, **PS1**, **PS2**, **IFS**.

SEE ALSO

env(1), login(1), sh(1), exec(2), getenv(3C), profile(4), term(5).

NAME

eqnchar – special character definitions for eqn and neqn

SYNOPSIS

eqn /usr/pub/eqnchar [files] | **troff** [options]

neqn /usr/pub/eqnchar [files] | **nroff** [options]

DESCRIPTION

Eqnchar contains *troff*(1) and *nroff*(1) character definitions for constructing characters that are not available on the Wang Laboratories, Inc. C/A/T phototypesetter. These definitions are primarily intended for use with *eqn*(1) and *neqn*; *eqnchar* contains definitions for the following characters:

ciplus	*ciplus*	||	||	*square*	*square*
citimes	*citimes*	*langle*	*langle*	*circle*	*circle*
wig	*wig*	*rangle*	*rangle*	*blot*	*blot*
– wig	*– wig*	*hbar*	*hbar*	*bullet*	*bullet*
> wig	*> wig*	*ppd*	*ppd*	*prop*	*prop*
< wig	*< wig*	<- >	<→	*empty*	*empty*
= wig	*=wig*	<=>	≤>	*member*	*member*
star	*star*	|<	|<	*nomem*	*nomem*
bigstar	*bigstar*	|>	|>	*cup*	*cup*
= dot	*=dot*	*ang*	*ang*	*cap*	*cap*
orsign	*orsign*	*rang*	*rang*	*incl*	*incl*
andsign	*andsign*	*3dot*	*3dot*	*subset*	*subset*
= del	*=del*	*thf*	*thf*	*supset*	*supset*
oppA	*oppA*	*quarter*	*quarter*	*!subset*	*!subset*
oppE	*oppE*	*3quarter*	*3quarter*	*!supset*	*!supset*
angstrom	*angstrom*	*degree*	*degree*	*scrL*	*scrL*
==<	⟸	==>	⟹		

FILES

/usr/pub/eqnchar

SEE ALSO

eqn(1), nroff(1), troff(1).

NAME

fcntl - file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This #include file describes *requests* and *arguments* to *fcntl*(2) and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/* (The first three can only be set by open) */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2
#define O_NDELAY 04      /* Non-blocking I/O */
#define O_APPEND 010     /* append (writes guaranteed at the end) */

/* Flag values accessible only to open(2) */
#define O_CREAT  00400   /* open with file (uses third arg)*/
#define O_TRUNC  01000   /* open with truncation */
#define O_EXCL   02000   /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD  0       /* Duplicate fildes */
#define F_GETFD  1       /* Get fildes flags */
#define F_SETFD  2       /* Set fildes flags */
#define F_GETFL  3       /* Get file flags */
#define F_SETFL  4       /* Set file flags */
```

SEE ALSO

fcntl(2), open(2).

NAME

greek – graphics for the extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek [| **greek** – **T**terminal]

DESCRIPTION

Greek gives the mapping from ASCII to the "shift-out" graphics in effect between SO and SI on TELETYPE Model 37 terminals equipped with a 128-character type-box. These are the default greek characters produced by *nroff*. The filters of *greek*(1) attempt to print them on various other terminals. The file contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ϵ	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	$\neg$	_
partial	∂	]	integral	$\int$	^			

FILES

/usr/pub/greek

SEE ALSO

300(1), 4014(1), 450(1), greek(1), hp(1), tc(1), nroff(1).

NAME

mailaddr – mail addressing description

DESCRIPTION

Mail addresses are based on the ARPANET protocol listed at the end of this manual page. These addresses are in the general format

user@domain

where a domain is a hierarchical dot separated list of subdomains. For example, the address

eric@monet.Berkeley.ARPA

is normally interpreted from right to left: the message should go to the ARPA name tables (which do not correspond exactly to the physical ARPANET), then to the Berkeley gateway, after which it should go to the local host monet. When the message reaches monet it is delivered to the user "eric".

Unlike some other forms of addressing, this does not imply any routing. Thus, although this address is specified as an ARPA address, it might travel by an alternate route if that was more convenient or efficient. For example, at Berkeley the associated message would probably go directly to monet over the Ethernet rather than going via the Berkeley ARPANET gateway.

Abbreviation. Under certain circumstances it may not be necessary to type the entire domain name. In general anything following the first dot may be omitted if it is the same as the domain from which you are sending the message. For example, a user on "calder.Berkeley.ARPA" could send to "eric@monet" without adding the ".Berkeley.ARPA" since it is the same on both sending and receiving hosts.

Certain other abbreviations may be permitted as special cases. For example, at Berkeley ARPANET hosts can be referenced without adding the ".ARPA" as long as their names do not conflict with a local host name.

Compatibility. Certain old address formats are converted to the new format to provide compatibility with the previous mail system. In particular,

host:user

is converted to

user@host

to be consistent with the *rcp*(1C) command.

Also, the syntax:

host!user

is converted to:

user@host.UUCP

This is normally converted back to the "host!user" form before being sent on for compatibility with older UUCP hosts.

The current implementation is not able to route messages automatically through the UUCP network. Until that time you must explicitly tell the mail system which hosts to send your message through to get to your final destination.

Case Distinctions. Domain names (i.e., anything after the "@" sign) may be given in any mixture of upper and lower case with the exception of UUCP hostnames. Most hosts accept any mixture of case in user names, with the notable exception of MULTICS sites.

Differences with ARPA Protocols. Although the UNIX addressing scheme is based on the ARPA mail addressing protocols, there are some significant differences.

At the time of this writing the only "top level" domain defined by ARPA is the ".ARPA" domain itself. This is further restricted to having only one level of host specifier. That is, the only addresses that ARPA accepts at this time must be in the format "user@host.ARPA" (where "host" is one word). In particular, addresses such as:

eric@monet.Berkeley.ARPA

are not currently legal under the ARPA protocols. For this reason, these addresses are converted to a different format on output to the ARPANET, typically:

eric%monet@Berkeley.ARPA

Route-addrs. Under some circumstances it may be necessary to route a message through several hosts to get it to the final destination. Normally this routing is done automatically, but sometimes it is desirable to route the message manually. An address that shows these relays are termed "route-addrs." These use the syntax:

<@hosta,@hostb:user@hostc>

This specifies that the message should be sent to hosta, from there to hostb, and finally to hostc. This path is forced even if there is a more efficient path to hostc.

Route-addrs occur frequently on return addresses, since these are generally augmented by the software at each host. It is generally possible to ignore all but the "user@host" part of the address to determine the actual sender.

Postmaster. Every site is required to have a user or user alias designated "postmaster" to which problems with the mail system may be addressed.

CSNET. Messages to CSNET sites can be sent to "user.host@UDel-Relay".

BERKELEY

The following comments apply only to the Berkeley environment.

Host Names. Many of the old familiar host names are being phased out. In particular, single character names as used in Berknet are incompatible with the larger world of which Berkeley is now a member. For this reason the following names are being obsoleted. You should notify any correspondents of your new address as soon as possible.

OLD	NEW			
p	ucbcad	j	ingvax	ucbingres
v csvax	ucbernie	r	arpavax	ucbarpa
n	ucbkim	y		ucbcory

The old addresses will be rejected as unknown hosts sometime in the near future.

What's My Address? If you are on a local machine, say monet, your address is

yourname@monet.Berkeley.ARPA

However, since most of the world does not have the new software in place yet, you will have to give correspondents slightly different addresses. From the ARPANET, your address would be:

yourname%monet@Berkeley.ARPA

From UUCP, your address would be:

ucbvax!yourname%monet

Computer Center. The Berkeley Computer Center is in a subdomain of Berkeley. Messages to the computer center should be addressed to:

user%host.CC@Berkeley.ARPA

The alternate syntax:

user@ host.CC

may be used if the message is sent from inside Berkeley.

For the time being Computer Center hosts are known within the Berkeley domain, i.e., the ".CC" is optional. However, it is likely that this situation will change with time as both the Computer Science department and the Computer Center grow.

Bitnet. Hosts on bitnet may be accessed using:

user@ host.BITNET

SEE ALSO

mail(1), sendmail(8); Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages,* RFC822.

NAME

man – macros for formatting entries in this manual

SYNOPSIS

nroff – man files

troff – man [**– rs1**] files

DESCRIPTION

These *troff*(1) macros are used to lay out the format of the entries of this manual. A skeleton entry may be found in the file **/usr/man/u_man/man0/skeleton**. These macros are used by the *man*(1) command.

The default page size is 8.5″×11″, with a 6.5″×10″ text area; the **– rs1** option reduces these dimensions to 6″×9″ and 4.75″×8.375″, respectively; this option, which is not effective in *nroff*(1), also reduces the default type size from 10-point to 9-point and the vertical line spacing from 12-point to 10-point. The **– rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters and the page length to 84 lines, and it inhibits underlining; this option should not be confused with the **– Tvp** option of the *man*(1) command, which is available at some UNIX System sites.

Any *text* argument below may be one to six "words". Double quotes (**""**) must be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, .I may be used to italicize a whole line, or .SM followed by .B to make small bold text. By default, hyphenation is turned off for *nroff* but remains on for *troff*.

Type font and size are reset to default values before each paragraph and after processing font-setting and size-setting macros, e.g., **.I**, **.RB**, **.SM**. Tab stops are neither used nor set by any macro except **.DT** and **.TH**.

Default units for indents (*in*) are ens. When a macro is given without the *in* argument, the previous indent is used. The "remembered" indent is set to its default value by the **.TH**, **.P**, **.SH**, and **.SS** macros. This value is 7.2 ens in *troff* and 5 ens in *nroff*; both are equal to 0.5 inches in the default page size. This means that within each subheading section (SYNOPSIS, DESCRIPTION, etc.) the default left margin is 0.5 inches to the right of the page offset (i.e., normal left margin) of the page. If the entire page width is needed (e.g., to format a large table), use **.in** alone on a line to override the default indented margin.

Each macro description below includes the effect on indentation, as applicable.

.TH *t s c n* Set the title and entry heading; *t* is the title, *s* is the section number, *c* is extra commentary, e.g., "local", *n* is new manual name. Invokes **.DT** (see below).

.SH *text* Place subhead *text*, e.g., SYNOPSIS, here. The text lines that follow the heading are block-style paragraphs; the whole block is indented 0.5 inches.

.SS *text* Place sub-subhead *text*, e.g., **Options**, here. The text lines that follow the heading are block-style paragraphs; the whole block is indented 0.5 inches.

.B *text* Make *text* bold.

.I *text* Make *text* italic.

.SM *text* Make *text* 1 point smaller than default point size.

.RI *a b* Concatenate roman *a* with italic *b*, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold:

.IR .RB .BR .IB .BI

.P Skip one vertical space and begin a paragraph with normal font, point size, and indent (0.5 inches). .PP has the same effect as .P.

.HP *in* Skip one vertical space and begin a paragraph with a hanging indent. The first line of the paragraph will be indented the default 0.5 inches from the page offset. The other lines will be indented the additional number of ens specified by *in*.

.TP *in* Skip one vertical space and begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. The indentation from the beginning of the tag to the beginning of the paragraph is specified by the *in* argument. If the tag does not fit, it is printed on a separate line. Format within the paragraph can be controlled by using the *nroff* commands **.br** and **.nf** (refer to the *Document Processing Guide*).

.IP *t in* Same as **.TP** *in* with tag *t*; often used to get an indented paragraph without a tag.

.RS *in* Increase indentation relative to the current margin. If given without an argument, the text following the macro will be indented 0.5 inches. The **.RS** macro does not cause a vertical space to be inserted before the following output. Use **.sp** on the line before the **.RS** line to obtain this space. If an *in* argument is given, the **.RS** macro will indent the following output *in* units from the current left margin.

.RE *k* Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level. **.RS**/**.RE** pairs can be nested.

.PM *m* Produces proprietary markings; where *m* may be **P** for PRIVATE, **N** for NOTICE, **BP** for BELL LABORATORIES PROPRIETARY, or **BR** for BELL LABORATORIES RESTRICTED.

.DT Restore default tab settings (every 7.2 ens in *troff*, 5 ens in *nroff*).

.PD *v* Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (0.4v in *troff*, 1v in *nroff*).

The following *strings* are defined:

*R in *troff*, **(Reg.)** in *nroff*(1).

*S Change to default type size.

*(Tm Trademark indicator.

The following *number registers* are given default values by **.TH**:

IN Left margin indent relative to subheads (default is 7.2 ens in *troff*, 5 ens in *nroff*).

LL Line length including IN.

PD Current interparagraph distance.

WARNINGS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *troff* and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of),], and }, and *A* stands for any alphanumeric character.

If a manual entry needs to be preprocessed by *cw*(1), *eqn*(1) (or *neqn*), and/or *tbl*(1), it must begin with a special line (described in *man*(1)), causing the *man* command to invoke the appropriate preprocessor(s).

The programs that prepare the Table of Contents and the Permuted Index for the User's Manual and Administrator's Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

name[, name, name ...] \- explanatory text

To eliminate ambiguity, the macro package increases the inter-word spaces in the *SYNOPSIS* section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font by the constant-width font– see *cw*(1)). Of course, if the input text of an entry contains requests for other fonts (e.g., .I, .RB, \fI), the corresponding fonts must be mounted. If a single word or short phrase needs to be italicized or emboldened, the following usage can be placed within a line, rather than creating a separate .B or .I line: \fItext\fR.

Nroff and *troff* formatting commands and macros are described in the Document Processing Guide.

FILES

/usr/lib/tmac/tmac.an
/usr/lib/macros/cmp.[nt].[dt].an
/usr/lib/macros/ucmp.[nt].an
/usr/man/[ua]_man/man0/skeleton

SEE ALSO

man(1), nroff(1), troff(1).

BUGS

When using the macros to alternate fonts (e.g., .RB, .IR), quotation marks must be used to maintain spacing. For example, **.IR filename** produces filename as one word. **.IR "file " name** produces it as two words.

NAME

mm – the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff – mm [options] [files]

nroff – cm [options] [files]

mmt [options] [files]

troff – mm [options] [files]

troff – cm [options] [files]

DESCRIPTION

This package provides a formatting capability for a wide variety of documents. The manner in which a document is typed and edited is essentially independent of whether the document is to be eventually formatted at a terminal or phototypeset. See the references below for further details.

The **– mm** option causes *nroff*(1) and *troff*(1) to use the non-compacted version of the macro package, while the **– cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/mm[nt]	non-compacted version of the package
/usr/lib/macros/cmp.[nt].[dt].m	compacted version of the package
/usr/lib/macros/ucmp.[nt].m	initializers for the compacted version of the package

SEE ALSO

mm(1), mmt(1), nroff(1), troff(1).

Document Processing Guide.

MM– Memorandum Macros by D. W. Smith and J. R. Mashey.

Typing Documents with MM by D. W. Smith and E. M. Piskorik.

NAME

mosd – the OSDD adapter macro package for formatting documents

SYNOPSIS

osdd [options] [files]

mm – mosd [options] [files]

nroff – mm – mosd [options] [files]

nroff – cm – mosd [options] [files]

mmt – mosd [options] [files]

troff – mm – mosd [options] [files]

troff – cm – mosd [options] [files]

DESCRIPTION

The OSDD adapter macro package is a tool used in conjunction with the *mm*(1) macro package to prepare Operations Systems Deliverable Documentation. Many of the OSDD Standards are different than the default format provided by *mm*(1). The OSDD adapter package sets the appropriate *mm*(1) options for automatic production of the OSDD Standards. The OSDD adapter package also generates the correct OSDD page headers and footers, heading styles, Table of Contents format, etc.

OSDD document (input) files are prepared with the *mm*(1) macros. Additional information which must be given at the beginning of the document file is specified by the following string definitions:

```
.ds H1 document-number
.ds H2 section-number
.ds H3 issue-number
.ds H4 date
.ds H5 rating
```

The *document-number* should be of the standard 10-character format. The words "Section" and "Issue" should not be included in the string definitions; they will be supplied automatically when the document is printed. For example,

```
.ds H1 OPA– 1P135– 01
.ds H2 4
.ds H3 2
```

automatically produces

```
OPA-1P135-01
Section 4
Issue 2
```

as the document page header. Quotation marks are not used in string definitions.

If certain information is not to be included in a page header, the string is defined as null; e.g.,

```
.ds H2
```

means that there is no *section-number*.

The OSDD Standards require that the *Table of Contents* be numbered beginning with *Page 1*. By default, the first page of text will be numbered *Page 2*. If the *Table of Contents* has more than one page, for example *n*, either – **rP***n+1* must be included as a command line option or **.nr P n** must be included in the document file. For example, if the *Table of Contents* is four pages, use – **rP5** on the command line or **.nr P 4** in the document file.

The OSDD Standards require that certain information such as the document *rating* appear on the *Document Index* or on the *Table of Contents* page if there is no index. By default, it is assumed that an index has been prepared separately. If there is no index, the following must be

included in the document file:

```
.nr Di 0
```

This will ensure that the necessary information is included on the *Table of Contents* page.

The OSDD Standards require that all numbered figures be placed at the end of the document. The **.Fg** macro is used to produce full page figures. This macro produces a blank page with the appropriate header, footer, and figure caption. Insertion of the actual figure on the page is a manual operation. The macro usage is

```
.Fg page-count "figure caption"
```

where *page-count* is the number of pages required for a multi-page figure (default 1 page).

Figure captions are produced by the **.Fg** macro using the .BS/.BE macros; therefore, the .BS/.BE macros are not available for users. The **.Fg** macro cannot be used within the document unless the final **.Fg** in a series of figures is followed by a .SK macro to force out the last figure page.

The *Table of Contents* for OSDD documents (see Figure 4 in Section 4.1 of the OSDD Standards) is produced with:

```
.Tc
System Type
System Name
Document Type
.Td
```

The **.Tc/.Td** macros are used instead of the **.TC** macro from *mm*(1).

By default, the adapter package causes the NOTICE disclosure statement to be printed. The **.PM** macro may be used to suppress the NOTICE or to replace it with the PRIVATE disclosure statement as follows:

```
.PM      none printed
.PM P    PRIVATE printed
.PM N    NOTICE printed (default)
```

The .P macro is used for paragraphs. The **Np** register is set automatically to indicate the paragraph numbering style. It is very important that the .P macro be used correctly. All paragraphs (including those immediately following a .H macro) must use a .P macro. Unless there is a .P macro, there will not be a number generated for the paragraph. Similarly, the .P macro should not be used for text which is not a paragraph. The .SP macro may be appropriate for these cases, e.g., for "paragraphs" within a list item.

The page header format is produced automatically in accordance with the OSDD Standards. The OSDD Adapter macro package uses the **.TP** macro for this purpose. Therefore the **.TP** macro normally available in *mm*(1) is not available for users.

FILES

/usr/lib/tmac/tmac.osd

SEE ALSO

mm(1), mmt(1), nroff(1), troff(1), mm(5).

MM- Memorandum Macros by D. W. Smith and J. R. Mashey.

Operations Systems Deliverable Documentation Standards, June 1980.

NAME

mptx – the macro package for formatting a permuted index

SYNOPSIS

nroff – mptx [options] [files]

troff – mptx [options] [files]

DESCRIPTION

This package provides a definition for the **.xx** macro used for formatting a permuted index as produced by *ptx*(1). This package does not provide any other formatting capabilities such as headers and footers. If these or other capabilities are required, the *mptx* macro package may be used in conjunction with the *mm*(1) macro package. In this case, the **– mptx** option must be invoked *after* the **– mm** call. For example:

nroff – cm – mptx *file*

or

mm – mptx *file*

FILES

/usr/lib/tmac/tmac.ptx	pointer to the non-compacted version of the package
/usr/lib/macros/ptx	non-compacted version of the package

SEE ALSO

mm(1), nroff(1), ptx(1), troff(1), mm(5).

NAME

mv – a troff macro package for typesetting viewgraphs and slides

SYNOPSIS

mvt [**– a**] [options] [files]

troff [**– a**] [**– rX1**] **– mv** [options] [files]

DESCRIPTION

This package makes it easy to typeset viewgraphs and projection slides in a variety of sizes. A few macros (briefly described below) accomplish most of the formatting tasks needed in making transparencies. All the facilities of *troff*(1), *cw*(1), *eqn*(1), and *tbl*(1) are available for more difficult tasks.

The output can be previewed on most terminals (in particular, the Tektronix 4014) and on the Versatec printer. For these two devices, specify the **– rX1** option (this option is automatically specified by the *mvt* command when that command is invoked with the **– T4014** or **– Tvp** options; see *mmt*(1)). To preview output on other terminals, specify the **– a** option.

The available macros are:

.VS [*n*] [*i*] [*d*] Foil-start macro; foil size is to be 7″×7″; *n* is the foil number, *i* is the foil identification, *d* is the date; the foil-start macro resets all parameters (e.g., indent, point size) to initial default values, except for the values of *i* and *d* arguments inherited from a previous foil-start macro; it also invokes the **.A** macro (see below).

The naming convention for this and the following 8 macros is that the first character of the name (**V** or **S**) distinguishes between viewgraphs and slides, while the second character indicates whether the foil is square (**S**), small wide (**w**), small high (**h**), big wide (**W**), or big high (**H**). Slides are narrower than the corresponding viewgraphs: the ratio of the longer dimension to the shorter one is larger for slides than for viewgraphs. As a result, slide foils can be used for viewgraphs, but not vice versa; on the other hand, viewgraphs can accommodate a bit more text.

.Vw [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 7″ wide ×5″ high.
.Vh [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 5″×7″.
.VW [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 7″×5.4″.
.VH [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 7″×9″.
.Sw [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 7″×5″.
.Sh [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 5″×7″.
.SW [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 7″×5.4″.
.SH [*n*] [*i*] [*d*] Same as **.VS**, except that foil size is 7″×9″.
.A [*x*] Place text that follows at the first indentation level (left margin); the presence of *x* suppresses the line spacing from the preceding text.
.B [*m* [*s*]] Place text that follows at the second indentation level; text is preceded by a mark; *m* is the mark (default is a large bullet); *s* is the increment or decrement to the point size of the mark with respect to the *prevailing* point size (default is 0); if *s* is 100, it causes the point size of the mark to be the same as that of the *default* mark.
.C [*m* [*s*]] Same as **.B**, but for the third indentation level; default mark is a dash.
.D [*m* [*s*]] Same as **.B**, but for the fourth indentation level; default mark is a small bullet.
.T *string* *String* is printed as an over-size, centered title.
.I [*in*] [*a* [*x*]] Change the current text indent (does not affect titles); *in* is the indent (in inches unless dimensioned, default is 0); if *in* is signed, it is an increment

or decrement; the presence of *a* invokes the **.A** macro (see below) and passes *x* (if any) to it.

.S [*p*] [*l*] Set the point size and line length; *p* is the point size (default is "previous"); if *p* is 100, the point size reverts to the *initial* default for the current foil-start macro; if *p* is signed, it is an increment or decrement (default is 18 for **.VS**, **.VH**, and **.SH**, and 14 for the other foil-start macros); *l* is the line length (in inches unless dimensioned; default is 4.2″ for **.Vh**, 3.8″ for **.Sh**, 5″ for **.SH**, and 6″ for the other foil-start macros).

.DF *n f* [*n f*...] Define font positions; may not appear within a foil's input text (i.e., it may only appear after all the input text for a foil, but before the next foil-start macro); *n* is the position of font *f*; up to 4 "*n f*" pairs may be specified; the first font named becomes the *prevailing* font; the initial setting is (**H** is a synonym for **G**):

.DF 1 H 2 I 3 B 4 S

.DV [*a*] [*b*] [*c*] [*d*] Alter the vertical spacing between indentation levels; *a* is the spacing for **.A**, *b* is for **.B**, *c* is for **.C**, and *d* is for **.D**; all non-null arguments must be dimensioned; null arguments leave the corresponding spacing unaffected; initial setting is:

.DV .5v .5v .5v 0v

.U *str1* [*str2*] Underline *str1* and concatenate *str2* (if any) to it.

The last 4 macros in the above list do not cause a break; the **.I** macro causes a break only if it is invoked with more than one argument; all the other macros cause a break.

The macro package also recognizes the following upper-case synonyms for the corresponding lower-case *troff* requests:

.AD .BR .CE .FI .HY .NA .NF .NH .NX .SO .SP .TA .TI

The **Tm** string produces the trademark symbol.

The input tilde (~) character is translated into a blank on output.

See the references cited below for further details.

FILES

/usr/lib/tmac/tmac.v
/usr/lib/macros/vmca

SEE ALSO

cw(1), eqn(1), mmt(1), tbl(1), troff(1).
Document Processing Guide.
A Macro Package for View Graphs and Slides by T. A. Dolotta and D. W. Smith.

BUGS

The **.VW** and **.SW** foils are meant to be 9″ wide by 7″ high, but because the typesetter paper is generally only 8″ wide, they are printed 7″ wide by 5.4″ high and have to be enlarged by a factor of 9/7 before use as viewgraphs; this makes them less useful.

NAME

regexp – regular expression compile and match routines

SYNOPSIS

```
#define INIT <declarations>
#define GETC( ) <getc code>
#define PEEKC( ) <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf

int step(string, expbuf)
char *string, *expbuf;
```

DESCRIPTION

This page describes general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), and *expr*(1), which perform regular expression matching, use this source file. Therefore, only the *regexp* file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following 5 macros declared before the **#include <regexp.h>** statement. These macros are used by the *compile* routine.

GETC( ) — Return the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.

PEEKC( ) — Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).

UNGETC(*c*) — Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.

RETURN(*pointer*) — This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

ERROR(*val*) — This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

compile(instring, expbuf, endbuf, eof)

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf– expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character that marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each program that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

step(string, expbuf)

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero, if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points to the character after the last character that matches the regular expression. Thus, if the regular expression matches the entire line, loc1 will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set, *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance*; *step* continues until *advance* returns a one indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line); for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC( )         (*sp++)
#define PEEKC( )        (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr( )

#include <regexp.h>
...
        compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
        if(step(linebuf, expbuf))
                succeed( );
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.

NAME

stat – data returned by stat system call

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
```

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
/*
 * Structure of the result of stat
 */

struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};

#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFIFO  0010000 /* fifo */
#define S_ISUID  04000   /* set user id on execution */
#define S_ISGID  02000   /* set group id on execution */
#define S_ISVTX  01000   /* save swapped text even after use */
#define S_IREAD  00400   /* read permission, owner */
#define S_IWRITE 00200   /* write permission, owner */
#define S_IEXEC  00100   /* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), types(5).

NAME

term – conventional names for terminals

DESCRIPTION

The names in this file are used by certain commands (e.g., *nroff*, *mm*(1), *man*(1), *tabs*(1)) and are maintained as part of the shell environment (see *sh*(1), *profile*(4), and *environ*(5)) in the variable $TERM:

1520	Datamedia 1520
155	Motorola EXORterm 155
1620	Diablo 1620 and others using the HyType II printer
1620–12	same, in 12-pitch mode
165	Motorola EXORset 165
2621	Hewlett-Packard HP2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer - compressed mode
2631–e	Hewlett-Packard 2631 line printer - expanded mode
2640	Hewlett-Packard HP2640 series
2645	Hewlett-Packard HP264n series (other than the 2640 series)
300	DASI/DTC/GSI 300 and others using the HyType I printer
300–12	same, in 12-pitch mode
300s	DASI/DTC/GSI 300s
382	DTC 382
300s–12	same, in 12-pitch mode
3045	Datamedia 3045
33	TELETYPE Terminal Model 33 KSR
37	TELETYPE Terminal Model 37 KSR
40–2	TELETYPE Terminal Model 40/2
40–4	TELETYPE Terminal Model 40/4
4540	TELETYPE Terminal Model 4540
3270	IBM Model 3270
4000a	Trendata 4000a
4014	Tektronix 4014
43	TELETYPE Model 43 KSR
450	DASI 450 (same as Diablo 1620)
450–12	same, in 12-pitch mode
735	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
sync	generic name for synchronous TELETYPE 4540-compatible terminals
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
tvi950	TeleVideo 950

Local changes to this list are common. Refer to **/etc/termcap** for information on terminals supported for your system.

Up to 8 characters, chosen from [– a– z0– 9], make up a basic terminal name. Terminal submodels and operational modes are distinguished by suffixes beginning with a – . Names should be based on original vendors, rather than local distributors. A terminal acquired from one

vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept arguments of the form **–T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable $TERM, which, in turn, should contain *term*.

SEE ALSO

mm(1), nroff(1), tplot(1G), sh(1), stty(1), tabs(1), profile(4), environ(5).

BUGS

Programs that should make use of this file do not adhere to the nomenclature in a consistent manner.

NAME

intro - introduction to games

DESCRIPTION

This section describes the recreational and educational programs found in the directory **/usr/games**. The availability of these programs may vary from system to system.

NAME

arithmetic – provide drill in number facts

SYNOPSIS

/usr/games/arithmetic [+− x/] [range]

DESCRIPTION

Arithmetic types out simple arithmetic problems, and waits for an answer to be typed in. If the answer is correct, it types back "Right!", and a new problem. If the answer is wrong, it replies "What?", and waits for another answer. Every twenty problems, it publishes statistics on correctness and the time required to answer.

To quit the program, type an interrupt (delete).

The first optional argument determines the kind of problem to be generated; +, − , x, and / respectively cause addition, subtraction, multiplication, and division problems to be generated. One or more characters can be given; if more than one is given, the different types of problems will be mixed in random order; default is +− .

Range is a decimal number; all addends, subtrahends, differences, multiplicands, divisors, and quotients will be less than or equal to the value of *range*. Default *range* is 10.

At the start, all numbers less than or equal to *range* are equally likely to appear. If the respondent makes a mistake, the numbers in the problem which was missed become more likely to reappear.

As a matter of educational philosophy, the program will not give correct answers, since the learner should, in principle, be able to calculate them. Thus the program is intended to provide drill for someone just past the first learning stage, not to teach number facts *de novo*. For almost all users, the relevant statistic should be time per problem, not percent correct.

NAME

back – the game of backgammon

SYNOPSIS

/usr/games/back

DESCRIPTION

Back is a program which provides a partner for the game of backgammon. It is designed to play at three different levels of skill, one of which you must select. In addition to selecting the opponent's level, you may also indicate that you would like to roll your own dice during your turns (for the superstitious players). You will also be given the opportunity to move first. The practice of each player rolling one die for the first move is not incorporated.

The points are numbered 1– 24, with 1 being white's extreme inner table, 24 being brown's inner table, 0 being the bar for removed white pieces and 25 the bar for brown. For details on how moves are expressed, type **y** when *back* asks "Instructions?" at the beginning of the game. When *back* first asks "Move?", type **?** to see a list of move options other than entering your numerical move.

When the game is finished, *back* will ask you if you want the log. If you respond with **y**, *back* will attempt to append to or create a file **back.log** in the current directory.

FILES

/usr/games/lib/backrules	rules file
/tmp/b*	log temp file
back.log	log file

BUGS

The only level really worth playing is "expert", and it only plays the forward game.
Back will complain loudly if you attempt to make too *many* moves in a turn, but will become very silent if you make too *few*.
Doubling is not implemented.

NAME

bj – the game of black jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

Bj is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is $2 every hand.

A player "natural" (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a "push" (no money exchange).

If the dealer has an ace up, the player is allowed to make an "insurance" bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.

If the player is dealt two cards of the same value, he is allowed to "double". He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)

If a dealt hand has a total of ten or eleven, the player may "double down". He may double the bet ($2 to $4) and receive exactly one more card on that hand.

Under normal play, the player may "hit" (draw a card) as long as his total is not over twenty-one. If the player "busts" (goes over twenty-one), the dealer wins the bet.

When the player "stands" (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.

If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by **y** followed by a new-line for "yes", or just new-line for "no".

```
?               (means, "do you want a hit?")
Insurance?
Double down?
```

Every time the deck is shuffled, the dealer so states and the "action" (total bet) and "standing" (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

NAME

craps – the game of craps

SYNOPSIS

/usr/games/craps

DESCRIPTION

Craps is a form of the game of craps that is played in Las Vegas. The program simulates the *roller*, while the user (the *player*) places bets. The player may choose, at any time, to bet with the roller or with the *House*. A bet of a negative amount is taken as a bet with the House, any other bet is a bet with the roller.

The player starts off with a "bankroll" of $2,000.

The program prompts with:

bet?

The bet can be all or part of the player's bankroll. Any bet over the total bankroll is rejected and the program prompts with **bet?** until a proper bet is made.

Once the bet is accepted, the roller throws the dice. The following rules apply (the player wins or loses depending on whether the bet is placed with the roller or with the House; the odds are even). The *first* roll is the roll immediately following a bet:

1. On the first roll:

7 or 11	wins for the roller;
2, 3, or 12	wins for the House;
any other number	is the *point*, roll again (Rule 2 applies).

2. On subsequent rolls:

point	roller wins;
7	House wins;
any other number	roll again.

If a player loses the entire bankroll, the House will offer to lend the player an additional $2,000. The program will prompt:

marker?

A **yes** (or **y**) consummates the loan. Any other reply terminates the game.

If a player owes the House money, the House reminds the player, before a bet is placed, how many markers are outstanding.

If, at any time, the bankroll of a player who has outstanding markers exceeds $2,000, the House asks:

Repay marker?

A reply of **yes** (or **y**) indicates the player's willingness to repay the loan. If only 1 marker is outstanding, it is immediately repaid. However, if more than 1 marker are outstanding, the House asks:

How many?

markers the player would like to repay. If an invalid number is entered (or just a carriage return), an appropriate message is printed and the program will prompt with **How many?** until a valid number is entered.

If a player accumulates 10 markers (a total of $20,000 borrowed from the House), the program informs the player of the situation and exits.

Should the bankroll of a player who has outstanding markers exceed $50,000, the *total* amount of money borrowed will be *automatically* repaid to the House.

Any player who accumulates $100,000 or more breaks the bank. The program then prompts:

New game?

to give the House a chance to win back its money.

Any reply other than **yes** is considered to be a **no** (except in the case of **bet?** or **How many?**). To exit, send an interrupt (break), DEL, or control-D. The program will indicate whether the player won, lost, or broke even.

MISCELLANEOUS

The random number generator for the die numbers uses the seconds from the time of day. Depending on system usage, these numbers, at times, may seem strange but occurrences of this type in a real dice situation are not uncommon.

NAME

hangman – guess the word

SYNOPSIS

/usr/games/hangman [arg]

DESCRIPTION

Hangman chooses a word at least seven letters long from a dictionary. The user is to guess letters one at a time.

The optional argument *arg* names an alternate dictionary.

FILES

/usr/lib/w2006

BUGS

Hyphenated compounds are run together.

NAME

maze – generate a maze

SYNOPSIS

/usr/games/maze

DESCRIPTION

Maze asks a few questions and then prints a maze.

BUGS

Some mazes (especially small ones) have no solutions.

NAME

moo – guessing game

SYNOPSIS

/usr/games/moo

DESCRIPTION

Moo is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A "cow" is a correct digit in an incorrect position. A "bull" is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

NAME

ttt, cubic – tic-tac-toe

SYNOPSIS

/usr/games/ttt
/usr/games/cubic

DESCRIPTION

Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

Cubic plays three-dimensional tic-tac-toe on a 4×4×4 board. Moves are specified as a sequence of three coordinate numbers in the range 1-4.

FILES

/usr/games/ttt.k learning file

BUGS

Cubic does not yet work on VAX.

NAME

wump – the game of hunt-the-wumpus

SYNOPSIS

/usr/games/wump

DESCRIPTION

Wump plays the game of "Hunt the Wumpus." A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company*, 2, 2 (November 1973).

BUGS

It will never replace Adventure.

PERMUTED INDEX