In the above example, the environment variable MAKEINC will be expanded and used as the directory where the file Makepre.h and Makepost.h exist.

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.

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 new-lines 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

This feature is available only in the domestic (U.S.) version of the UNIX PC software. *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, its input and output will be pipes.

SEE ALSO

crypt(1), ed(1), passwd(4).

NAME

mesg – permit or deny messages

SYNOPSIS

mesg [**n**] [**y**]

DESCRIPTION

Mesg with argument **n** forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reports the current state without changing it.

FILES

/dev/tty*

SEE ALSO

write(1).

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

message – display error and help messages

SYNOPSIS

message [–**u**] [–**c**] [–**i**] text

DESCRIPTION

Message allows the shell programmer access to the *message*(3T) subroutine. *Text* is a text string with the standard special character conventions: **\n** for newline, etc.

The possible options are:

–**u**	Use the current window for the messages—resizes it to fit.
–**c**	Create a confirmation message (see MT_CONFIRM in *message*(3T)).
–**i**	Create a *pop-up* message—press any key to return to the caller (see MT_POPUP in *message*(3T)).

If no options are set, *message*(1) will generate an error message (see MT_ERROR in *message*(3T)).

EXAMPLES

The following example prints a confirmation message using the current window:

```
message -uc "Do you wish to continue"
        if [ "$?" != "0" ]
        then
                exit
        fi
```

SEE ALSO

message(3T), shform(1), tam(3T).

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

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* 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* and MM are generated, depending on the options selected.

Osdd is equivalent to the command **mm −mosd**.

Options for *mm* are given below. Any other arguments or flags (e.g., **−rC3**) are passed to *nroff* 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* will use 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* will use **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*.

−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 file names. (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 file name is given, standard input is read.

HINTS

1. *Mm* invokes *nroff* with the **–h** flag. With this flag, *nroff* assumes that the terminal has tabs set every 8 character positions.
2. Use the **–o***list* option of *nroff* 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* 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* (to stop between pages of output), use line-feed (rather than return or new-line) to restart the output. The **–s** option of *nroff* 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 will be printed on, you'll 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), nroff(1), tbl(1), profile(4), mm(5), term(5).
UNIX System Document Processing Guide.

DIAGNOSTICS

mm	"mm: no input file" if none of the arguments is a readable file and *mm* is not used as a filter.
checkmm	"Cannot open *filename*" if file(s) is unreadable. The remaining output of the program is diagnostic of the source file.

NAME

mmt, mvt – typeset documents, view graphs, 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* (not included on the UNIX PC), as opposed to formatting it via *nroff*; *mmt* uses the MM macro package, while *mvt* uses the Macro Package for View Graphs 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* 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* 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 UNIX 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*.
–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 file names.

Mvt is just a link to *mmt*.

HINT

Use the **–o***list* option of *troff* 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* 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), profile(4), environ(5), mm(5).
UNIX System Document Processing Guide.

DIAGNOSTICS

"m[mv]t: no input file" if 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 [**–cdflsu**] [*–n*] [*+linenumber*] [*+/pattern*] [name ...]

page *more options*

DESCRIPTION

More is a filter which allows examination of a continuous text one screen full (or window full) at a time on a soft-copy terminal. It normally pauses after each screen full, 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 screen full is displayed. Other possibilities 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.

–c *More* will draw each page by beginning at the top of the screen and erasing each line just before it draws on it. This avoids scrolling the screen, making it easier to read while *more* is writing. This option will be ignored if the terminal does not have the ability to clear to the end of a line.

–d *More* will prompt the user with the message *Hit space to continue, Rubout to abort* at the end of each screen full. 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 lines, 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 positions, 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 screen full had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

–s Squeeze multiple blank lines from the output, producing only one blank line. Especially helpful when viewing *nroff* output, this option maximizes the useful information present on the screen.

–u Normally, *more* will handle underlining such as produced by *nroff* in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a stand-out mode, *more* will output appropriate escape sequences to enable underlining or stand-out mode for underlined information in the source file. The *-u* option suppresses this processing.

+*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 screen full is printed (but only if a full screen is being printed), and *k* − 1 rather than *k* − 2 lines are printed in each screen full, where *k* is the number of lines the terminal can display.

More looks in the *TERMCAP* environment variable or 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.

More looks in the environment variable *MORE* to pre-set any flags desired. For example, if you prefer to view files using the *-c* mode of operation, the *csh* command *setenv MORE -c* or the *sh* command sequence *MORE='-c' ; export MORE* would cause all invocations of *more*, including invocations by programs such as *man* and *msgs*, to use this mode. Normally, the user will place the command sequence which sets up the *MORE* environment variable in the **.cshrc** or **.profile** file.

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 screen full 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)
*i*z	same as typing a space except that *i*, if present, becomes the new window size.
*i*s	skip *i* lines and print a screen full of lines
*i*f	skip *i* screen fulls and print a screen full 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 screen full 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 -s
```

FILES

/etc/termcap	Terminal data base
/usr/lib/more.help	Help file

SEE ALSO

sh(1), environ(5).

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 "–e15 –l60" will yield results different from "–l60 –e15". 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). In addition, *tabspec* may be ––, in which *newform* assumes that the tab specification is to be found in the first line read from the standard input (see *fspec*(4)). If no *tabspec* is given, *tabspec* defaults to –**8**. A *tabspec* of –**0** expects no tabs; if any are found, they are treated as –**1**.

–**o***tabspec* Output tab specification: replaces spaces by tabs, according to the tab specifications given. The tab specifications are the same as for –**i***tabspec*. If no *tabspec* is given, *tabspec* defaults to –**8**. A *tabspec* of –**0** means that no spaces will be converted to tabs on output.

–**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 file-name
```

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 tab specification format line which is printed will correspond to the format specified in the *last* **−o** option. If no **−o** option is specified, the line which is printed will contain the default specification of **−8**.
−s	Shears off leading characters on each line up to the first tab and places up to 8 of the sheared characters at the end of the line. If more than 8 characters (not counting the first tab) are sheared, the eighth character is replaced by a * and any characters to the right of it are discarded. The first tab is always discarded.

An error message and program exit will occur if this option is used on a file without a tab on each line. The characters sheared off are saved internally until all other options specified are applied to that line. The characters are then added at the end of the processed line.

For example, to convert a file with leading digits, one or more tabs, and text on each line, to a file beginning with the text, all tabs after the first expanded to spaces, padded with spaces out to column 72 (or truncated to column 72), and the leading digits placed starting at column 73, the command would be:

```
newform −s −i −l −a −e file-name
```

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* will keep track of backspaces in order to line up tabs in the appropriate logical columns.

Newform will not prompt the user if a *tabspec* is to be read from the standard input (by use of **−i**−− or **−o**−−).

If the **−f** option is used, and the last **−o** option specified was **−o**−−, and was preceded by either a **−o**−− or a **−i**−−, the tab specification format line will be incorrect.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp [–] [group]

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login*(1M). 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(1M), 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

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 super-user 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 [**–h**type] [**–b**type] [**–f**type] [**–v**start#] [**–i**incr] [**–p**] [**–l**num] [**–s**sep] [**–w**width] [**–n**format] [**–d**delim] file

DESCRIPTION

Nl reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

Nl views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (e.g. no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line contents	*Start of*
\:\:\:	header
\:\:	body
\:	footer

Unless optioned otherwise, *nl* assumes the text being read is in a single logical page body.

Command options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

–b*type*	Specifies which logical page body lines are to be numbered. Recognized *types* and their meaning are: **a**, number all lines; **t**, number lines with printable text only; **n**, no line numbering; **p***string*, number only lines that contain the regular expression specified in *string*. Default *type* for logical page body is **t** (text lines numbered).
–h*type*	Same as **–b***type* except for header. Default *type* for logical page header is **n** (no lines numbered).
–f*type*	Same as **–b***type* except for footer. Default for logical page footer is **n** (no lines numbered).
–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
```

will number 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 [**–o**] [**–x**] [**–h**] [**–v**] [**–n**] [**–e**] [**–f**] [**–u**] [**–V**] [**–T**] file-names

DESCRIPTION

The *nm* command displays the symbol table of each common object file *file-name*. *File-name* may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, the following information will be printed:

Name	The name of the symbol.
Value	Its value expressed as an offset or an address depending on its storage class.
Class	Its storage class.
Type	Its type and derived type. If the symbol is an instance of a structure or of a union then the structure or union tag will be given following the type (e.g. struct-tag). If the symbol is an array, then the array dimensions will be given following the type (eg., **char**[n][m]). Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.
Size	Its size in bytes, if available. Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.
Line	The source line number at which it is defined, if available. Note that the object file must have been compiled with the **–g** option of the *cc*(1) command for this information to appear.
Section	For storage classes static and external, the object file section containing the symbol (e.g., text, data or bss).

The output of *nm* may be controlled using the following options:

–o	Print the value and size of a symbol in octal instead of decimal.
–x	Print the value and size of a symbol in hexadecimal instead of decimal.
–h	Do not display the output header data.
–v	Sort external symbols by value before they are printed.
–n	Sort external symbols by name before they are printed.
–e	Print only external and static symbols.
–f	Produce full output. Print redundant symbols (.text, .data and .bss), normally suppressed.
–u	Print undefined symbols only.

−V Print the version of the *nm* command executing on the standard error output.

−T By default, *nm* prints the entire name of the symbols listed. Since object files can have 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 its column, forcing every column after the name to be misaligned. The **−T** option causes *nm* to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **nm name −e −v** and **nm −ve name** print the static and external symbols sorted by value.

FILES

/usr/tmp/nm??????

CAVEATS

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"
if *name* cannot be read.

"nm: name: bad magic"
if *name* is not an appropriate common object file.

"nm: name: no symbols"
if the symbols have been stripped from *name*.

NAME

nohup – run a command immune to hangups and quits

SYNOPSIS

nohup command [arguments]

DESCRIPTION

Nohup executes *command* with hangups and quits ignored. If output is not re-directed by the user, it will be 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* cited below.

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* will halt *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 (new-lines 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 May 1979 Addendum to the *NROFF/ TROFF User's Manual* for details of compacting macro files).
–**T***name*	Prepare output for specified terminal. Known *names* 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
A TROFF Tutorial
col(1), eqn(1), greek(1), mm(1), tbl(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 −o*list* option inside a pipeline (e.g., with one or more of *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.

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* will be removed. Packed files can be restored to their original form using *unpack* or *pcat*.

Pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the – argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of – in place of *name* will cause the internal flag to be set and reset.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each **.z** file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Load modules, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

Pack returns a value that is the number of files that it failed to compress.

No packing will occur if:

> the file appears to be already packed;
> the file name has more than 12 characters;
> the file has links;
> the file is a directory;
> the file cannot be opened;
> no disk storage blocks will be saved by packing;
> a file called *name*.**z** already exists;
> the **.z** file cannot be created;
> an I/O error occurred during processing.

The last segment of the file name must contain no more than 12 characters to allow space for the appended **.z** extension. Directories cannot be compressed.

Pcat does for packed files what *cat*(1) does for ordinary files. The specified files are unpacked and written to the standard output. Thus to view a packed file named *name*.**z** use:

```
        pcat name.z
or just:
        pcat name
```

To make an unpacked copy, say *nnn*, of a packed file named *name*.**z** (without destroying *name*.**z**) use the command:

```
        pcat name >nnn
```

Pcat returns the number of files it was unable to unpack. Failure may occur if:

> the file name (exclusive of the **.z**) has more than 12 characters;
> the file cannot be opened;
> the file does not appear to be the output of *pack*.

Unpack expands files created by *pack*. For each file *name* specified in the command, a search is made for a file called *name*.**z** (or just *name*, if *name* ends in **.z**). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the **.z** suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

Unpack returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in *pcat*, as well as for the following:

> a file with the "unpacked" name already exists;
> if the unpacked file cannot be created.

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 super-user may change a password; the owner must prove he knows the old password. Only the super-user 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(1M), crypt(3C), passwd(4).

NAME

paste – merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste file1 file2 . . .
paste –**d** list file1 file2 . . .
paste –**s** [–**d** list] file1 file2 . . .

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). If you will, it 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 file name.

The meanings of the options are:

–**d** 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. This option allows replacing the *tab* character by one or more alternate characters (see below).

list One or more characters immediately following –**d** replace the default *tab* as the line concatenation character. The list is used circularly, i. e. when exhausted, it is reused. In parallel merging (i. e. no –**s** option), the lines from the last file are always terminated with a 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 –**d** option. Regardless of the *list*, the very last character of the file is forced to be a new-line.

– May be used in place of any file name, to read a line from the standard input. (There is no prompting).

EXAMPLES

ls | paste –d" " –	list directory in one column
ls | paste – – – –	list directory in four columns
paste –s –d"\t\n" file	combine pairs of lines into lines

SEE ALSO

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 **−s** option, no more than 12 input files may be specified.

NAME

path – locate executable file for command

SYNOPSIS

path *command*

DESCRIPTION

Path is a quick way to discover what executable file is behind a shell command. It searches each directory mentioned in your PATH environment variable until it finds an executable file called *command.*

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 below ***options*** 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 file name.
−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 five-line identifying header nor the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.
−s*c*	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

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* will 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 300 functions that may have call counters established during program execution. If this limit is exceeded, other data will be overwritten and the **mon.out** file will be corrupted. The number of call counters used will be reported automatically by the *prof* command whenever the number exceeds 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 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 or directory to be processed; non-SCCS files and unreadable files are silently ignored.

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

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

–**d**[*dataspec*]	Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *DATA KEYWORDS*) interspersed with optional user supplied text.
–**r**[*SID*]	Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.
–**e**	Requests information for all deltas created *earlier* than and including the delta designated via the –**r** keyletter.
–**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**.

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D* or *R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS: ...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS: ...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS: ...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
: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

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

TABLE 1 (Continued)

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

```
prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:"
-r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only keyletter argument allowed to be used with the *special case* is the **-a** keyletter.

FILES

/tmp/pr?????

SEE ALSO

admin(1), delta(1), get(1), help(1), sccsfile(4).
Source Code Control System User's Guide in the *UNIX System 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 information that is displayed 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** will cause the user block to be zeroed out.
–n *namelist*	The argument will be 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 from one another by a comma, or a list of terminal identifiers enclosed in double quotes and separated from one another 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 will be printed unless the **–f** option is used, in which case the login name will be 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 will 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 this datum.
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.
ADDR	(l)	The memory address of the process (a pointer to the segment table array on the 3B20S), if resident; otherwise, the disk address.
SZ	(l)	The size in blocks of the core image of the process.

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* 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* will reserve 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 (as to a page or chapter), separate from the text of the line. Attach that 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), 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 path name of the working (current) directory.

SEE ALSO

cd(1).

DIAGNOSTICS

"Cannot open .." and "Read error in .." indicate possible file system trouble and should be referred to a UNIX programming counselor.

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 on both execution time and program size. The command *regcmp* compiles the regular expressions in *file* and places the output in *file*.**i**. If the – option is used, the output will be placed in *file*.**c**. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotes. The output of *regcmp* is C source code. Compiled regular expressions are represented as **extern char** vectors. *File*.**i** files may thus be *included* 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*) will apply the regular expression named *abc* to *line*. Diagnostics are self-explanatory.

EXAMPLES

```
name    "([A-Za-z][A-Za-z0-9_]*)$0"

telno   "\({0,1}([2-9][01][1-9])$0\){0,1} *"
        "([2-9][0-9]{2})$1[ -]{0,1}"
        "([0-9]{4})$2"
```

In the C program that uses the *regcmp* output,

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named *telno* to *line*.

SEE ALSO

regcmp(3X).

NAME

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 was 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 antisocial 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 delta specified must *not* be that of a version being edited for the purpose of making a delta (i. e., if a *p-file* (see *get*(1)) exists for the named SCCS file, the delta specified 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 *UNIX System 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).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff −**r**SID1 −**r**SID2 [−**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 User's Guide
UNIX System User's Guide.

DIAGNOSTICS

"*file*: No differences"　　If the two versions are the same.
Use *help*(1) for explanations.

NAME

scrset – set screen save time

SYNOPSIS

scrset [*n*]

DESCRIPTION

Scrset enables and disables the screen save feature. When enabled, this feature causes the screen to go blank after a given interval of time has elapsed with no keyboard or mouse input; the next keystroke or mouse motion restores the screen display. This is a new feature of the UNIX PC 3.0 release.

The parameter *n*, if greater than 0, is the number of seconds to delay before turning off the screen. *N* equal to 0 turns off the screen save feature (this is the default condition). If *n* is less than 0, the screen is immediately turned off.

NAME

sdb – symbolic debugger

SYNOPSIS

sdb [–**w**] [–**W**] [objfil [corfil [directory-list]]]

DESCRIPTION

Sdb is a symbolic debugger that can be used with C 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* will be limited, but the file can still be examined and the program debugged. The default for *objfil* is **a.out**. *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core**. The core file need not be present. A – in place of *corfil* will force *sdb* to ignore any core image file. The colon separated list of directories (*directories-list*) is used to locate the source files used to build *objfil*.

It is useful to know that at any time there is a *current line* and *current file*. If *corfil* exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in *main*(). The current line and file may be changed with the source file examination commands.

Initially *sdb* has a greater-than character (>) prompt, which indicates that *sdb* is ready for the user to enter the first command. After *sdb* has begun, the prompt is <*x*>, where *x* is the name of the last command given.

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. Note that names in C are now of arbitrary length, *sdb* will no longer truncate names. Variables local to a procedure may be accessed using the form *procedure***:***variable*. If no procedure name is given, the procedure containing the current line is used by default.

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. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure referenced by *sdb*. An unqualified structure variable may also be used with various commands. Generally, *sdb* will interpret a structure as a set of variables. Thus, *sdb* will display the values of all the elements of a structure when it is requested to display a structure.

An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value *sdb* displays, not the addresses of individual elements.

Elements of a multidimensional array may be referenced as

> *variable*[*number*][*number*]...,

or as

> *variable*[*number*,*number*, ...].

In place of *number*, the form *number*;*number* may be used to indicate a range of values, * may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. As with structures, *sdb* displays all the values of an array or section of an array if trailing subscripts are omitted. It displays only the address of the array itself or section specified by the user if subscripts are omitted.

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 *filename*:*number* or *procedure*:*number*. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under *sdb*, all addresses refer to the executing program; otherwise they refer to *objfil* or *corfil*. An initial argument of **−w** permits overwriting locations in *objfil*.

Addresses.

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*) and the *file address* corresponding to a written *address* is calculated as follows:

$$b1 \leq address < e1$$

$$file\ address = address + f1 - b1$$

otherwise

$$b2 \leq address < e2$$

$$file\ address = address + f2 - b2$$

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 symbolically.
I	disassemble machine language instruction with addresses just printed numerically.

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 specified *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, both variables local to the current procedure and global variables are matched; if a procedure name is specified then only variables local to that procedure are matched. To match only global variables, the form **:***pattern* is used.

*linenumber***?***lm*
*variable***:?***lm*
Print the value at the address from **a.out** or I space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is 'i'.

*variable***=***lm*
*linenumber***=***lm*
*number***=***lm*
Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

*variable***!***value*
Set *variable* to the given *value*. The value may be a number, character constant or a variable. The value must be well defined; expressions which produce more than one value, such as structures, are not allowed. Character constants are denoted **'***character*. Numbers are viewed as integers unless a decimal point or exponent is used. In this case, they are treated as having the type double. Registers are viewed as integers. The *variable* may be an expression which 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 performing any type conversions necessary to perform the indicated assignment.

f Print the 68881 floating-point registers.

x Print the machine registers and the current machine-language instruction.

X Print the current machine-language instruction.

The commands for examining source files are:

e *procedure*
e *file-name*
e *directory/*

e *directory file-name*

The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure and file names 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 omitted.

?*regular expression*?

Search backward from the current line for a line containing a string matching *regular expression* as in *ed*(1). The trailing ? may be deleted.

p Print the current line.

z Print the current line followed by the next 9 lines. Set the current line to the last line printed.

w Window. Print the 10 lines around the current line.

number

Set the current line to the given line number. Print the new current line.

count+

Advance the current line by *count* lines. Print the new current line.

count−

Retreat the current line by *count* lines. Print the new current line.

The commands for controlling the execution of the source program are:

count **r** *args*
count **R**

Run the program with the given arguments. The **r** command with no arguments reuses the previous arguments to the program while the **R** command runs the program with no arguments. An argument beginning with < or > causes redirection for the standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber **c** *count*
linenumber **C** *count*

Continue after a breakpoint or interrupt. If *count* is given, it specifies the number of breakpoints to be ignored. **C** continues with the signal which caused the program to stop and **c** ignores it. If a *linenumber* is specified then a temporary breakpoint is placed at the line and execution is continued. This temporary 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 which 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 then execution stops just before the breakpoint and control is returned to *sdb*. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If **k** is used as a command to execute at a breakpoint, control returns to *sdb*, instead of continuing

execution.

B Print a list of the currently active breakpoints.

linenumber **d**
Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively: each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d** then the breakpoint is deleted.

D Delete all breakpoints.

l Print the last executed line.

linenumber **a**
Announce. If *linenumber* is of the form *proc***:***number*, the command effectively does a *linenumber* **b l**. If *linenumber* is of the form *proc***:**, the command effectively does a *proc***: b T**.

Miscellaneous commands:

!*command*
The command is interpreted by *sh*(1).

new-line
Perform the previous command again.

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, and then continue to accept commands from standard input. When *sdb* is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

M Print the address maps.

M [?/] [*] *b e f*
Record new values for the address map. The arguments ? and / specify the text and data maps respectively. The first segment, (*b1*,*e1*,*f1*), is changed unless * is specified, in which case the second segment (*b2*,*e2*,*f2*) of the mapping is changed. If fewer than three values are given, the remaining map parameters are left unchanged.

" *string*
Print the given string. The C escape sequences of the form \character are recognized, where *character* is a nonnumeric character.

q Exit the debugger.

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.

Sdb may be instructed to monitor a given memory location and stop the program when the value at that location changes in a given way. For example:

```
> if x < = 123
```

The above example instructs *sdb* to monitor the value at location *x*. When the user gives the command to continue (**c**), *sdb* checks the value of x at every source line executed and stops the program if the given condition becomes true. Note that use of this constraint slows the real-time execution of a program.

The syntax of the *if* command is as follows:

if Shows a list of the current data breakpoints; assigns a number to each.

if *var* Monitors the value of *var* and stops the program if the value changes. A variable name may be used for *var,* as well as a constant address. Comparisons are done as either 4-byte signed or 4-byte unsigned, depending on the data type. To perform a 1-byte or 2-byte comparison, an optional length value may accompany *var*. An example of a 2-byte comparison is

```
if x,2 = 0xff
```

if *var rel value*
Compares the value of *var* to the constant given and stops the program if the condition is true. The values of *rel* may be =, ==, <, <=, >, >=, or !=.

off *n* Disables or turns off a data breakpoint without removing it from the list.

on *n* Enables a breakpoint that was turned off.

out *n* Removes a breakpoint from the list.

Conditional breakpoints are used in a manner similar to data breakpoints, except that the user specifies a place in the program at which *sdb* should stop to check the data values. For example,

```
mysub:99 b if xyz = 123
```

The above example instructs *sdb* to check the value of xyz every time the program arrives at line 99 of subroutine *mysub*. If the condition is true, then execution stops there, as with a normal breakpoint. This type of breakpoint does not monitor the value xyz at every line of code, as the data breakpoint does.

FILES

a.out
core

SEE ALSO

cc(1), sh(1), a.out(4), core(4).

WARNINGS

When *sdb* prints the value of an external variable for which there is no debugging information, a warning is printed before the value. The value is assumed to be **int** (integer).

Data which are 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.

When setting a breakpoint at a procedure, *sdb* will inconsistently produce the incorrect line number. This seems to occur when the object file is newer than the source file. Recompiling the source program will correct this problem.

NAME

sdiff – side-by-side difference program

SYNOPSIS

sdiff [options ...] file1 file2

DESCRIPTION

Sdiff uses the output of *diff*(1) to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files is printed with a blank gutter between them if the lines are identical, a < in the gutter if the line only exists in *file1*, a > in the gutter if the line only exists in *file2*, and a | for lines that are different.

For example:

```
x     |     y
a           a
b     <
c     <
d           d
      >     c
```

The following options exist:

−w *n*	Use the next argument, *n*, as the width of the output line. The default line length is 130 characters.
−l	Only print the left side of any lines that are identical.
−s	Do not print identical lines.
−o *output*	Use the next argument, *output*, as the name of a third file that is created as a user controlled merging of *file1* and *file2*. Identical lines of *file1* and *file2* are copied to *output*. Sets of differences, as produced by *diff*(1), are printed; where a set of differences share a common gutter character. After printing each set of differences, *sdiff* prompts the user with a % and waits for one of the following user-typed commands:

l	append the left column to the output file
r	append the right column to the output file
s	turn on silent mode; do not print identical lines
v	turn off silent mode
e l	call the editor with the left column
e r	call the editor with the right column
e b	call the editor with the concatenation of left and right
e	call the editor with a zero length file
q	exit from the program

On exit from the editor, the resulting file is concatenated on the end of the *output* file.

SEE ALSO
diff(1), ed(1).

NAME

sed – stream editor

SYNOPSIS

sed [**–n**] [**–e** script] [**–f** 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** options, the flag **–e** may be omitted. The **–n** option suppresses the default output. A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation, *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a **D** command), applies in sequence all commands whose *addresses* select that pattern space, and at the end of the script copies the pattern space to the standard output (except under **–n**) and deletes the pattern space.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval.

An *address* is either a decimal number that counts input lines cumulatively across files, a $ that addresses the last line of input, or a context address, i.e., a */regular expression/* in the style of *ed*(1) modified thus:

In a context address, the construction \*?regular expression?*, where *?* is any character, is identical to */regular expression/*. Note that in the context address **\xabc\xdefx**, the second **x** stands for itself, so that the regular expression is **abcxdef**.

The escape sequence **\n** matches a 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).

NAME

setprint – send a different page length/width to an LP line printer

SYNOPSIS

setprint lines cols

DESCRIPTION

Lp uses a default page length (66 lines) and page width (132 columns) for printing. If the file to be printed has more than 132 columns, all characters beyond 132 would either be truncated or the printer would continue to print them all on the last character position.

Setprint allows you to change the line and column size parameters to whatever your printer can handle. However, *setprint* can only be used with a parallel line printer, and that printer must be online. Otherwise an I/O error will occur.

EXAMPLE

To change the page width to to 150 columns, use *setprint* as follows:

```
setprint  66 150
```

Use the following format to set the page width back to 132 columns:

```
setprint 66 132
```

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 but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

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 file-name generation (see *File Name 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 then substitute its value; otherwise substitute *word*.

${*parameter*:=*word*}
: If *parameter* is not set or is null then set it to *word*; the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${*parameter*:?*word*}
: If *parameter* is set and is non-null then substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, then the message "parameter null or not set" is printed.

${*parameter*:+*word*}
: If *parameter* is set and is non-null then substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is null:

```
echo ${d:- `pwd` }
```

If the colon (**:**) is omitted from the above expressions, 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*(1M)).

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.

File Name 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 file names that match the pattern. If no file name is found that matches the pattern, then the word is left unchanged. The character **.** at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

*	Matches any string, including the null string.
?	Matches any single character.
[. . .]	Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening ``['' is a "!" 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 then it is created; otherwise, it is truncated to zero length.
>>word	Use file *word* as standard output. If the file exists then 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 and command substitution occurs, (unescaped) \new-line is ignored, and \\ must be used to quote the characters \\, $, ‘, and the first character of *word*. If – is appended to <<, then all leading tabs are stripped from *word* and from the document.
<&digit	The standard input is duplicated from file descriptor *digit* (see *dup*(2)). Similarly for the standard output using >.
<&–	The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit, then 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 & then the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

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 any 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 / then the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell (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 then the exit status is that of the last command executed (an end-of-file will also cause the shell to exit.)

export [*name* . . .]
The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, then 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 *name*s are marked *readonly* and the values of these *name*s may not be changed by subsequent assignment. If no arguments are given, then a list of all *readonly* names is printed.

set [**−−ekntuvx** [*arg* . . .]]

−e	Exit immediately if a command exits with a nonzero 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 then 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/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 then 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 then 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, then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.
–r	If the **–r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

Rsh Only.

Rsh is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of *rsh* are identical to those of *sh*, except that the following are disallowed:

changing directory (see *cd*(1)),
setting the value of **$PATH,**
specifying path or command names containing /,
redirecting output (> and >>).

The restrictions above are enforced after **.profile** is interpreted.

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(1M), 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).

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

shform – displays menus and forms and returns user input to Bourne Shell procedures.

SYNOPSIS

RET = 'shform [-u] formname'

DESCRIPTION

The *shform* process displays a menu or form, waits for user input, and returns the result to the shell procedure.

Formname is a text document, called a form description file, that describes the menu or form to be displayed. Entries in the file use a keyword = value syntax. (The form and menu keywords are described below.) The file must be located in the */usr/lib/ua* directory. To insert a comment in the file, start the line with a pound sign (#).

The value returned by the file is stored in the shell variable *RET* as a list of words separated by spaces. If an error occurs, then $? will contain an error code.

-u causes *shform* to place its menu or form in the current window, resizing it appropriately to fix the menu or form. This option is recommended.

Shform returns the following exit codes:

0 – AOK
1 – Argument error
2 – Out of memory (malloc failed)
3 – Internal table overflow
4 – Syntax error in form description file

The words in $*RET* are as follows:

word 1 = Name of terminating key

if form, words 2 – n = Values of the form's fields

if menu, word 2 = Name of selected menu item

if multiselect menu, words 2 – n = Name of selected menu items

Form Definition Keywords

Form = form name
: Flags the start of a form. It is followed by a series of field definitions. The form name specified here is used as the title of the form. Only one Form keyword can be used in the file.

Name = field name
: Follows a Form keyword and starts a field definition. The field name specified here is used as the prompt for the field. The field name definition is followed by field attribute definitions:

Prompt = prompt string
: Displayed on the prompt line when the field is the current field.

Frow = number
    Defines the row in the form where the current field displays.

Ncol = number.
    Defines the column in the form where the field name displays.

Fcol = number
    Defines the column in the form where the field value displays.

Flen = number
    Defines the maximum length of the field value, in columns.

Value = initial field value
    Defines the initial contents (default value) for the field.

Rmenu = menu name
    If the field has an associated menu of options, this keyword is included. The menu name must be defined later in the file with the Menu keyword (see below).

Menuonly
    If this keyword is present then user editing of the field is forbidden and any key which is typed will cause the associated menu to display.

Menu Definition Keywords

Menu = menu name
    Begins a menu definition. When no form is defined, *shform* displays a menu instead of a form. In this case, only the first defined menu is displayed. If a form is defined, then the only menus displayed are those referenced in the form fields (via the Rmenu keyword, defined above). The Menu keyword is followed by a series of menu attribute definitions.

Prompt = prompt string
    The prompt string is displayed on the prompt line when the menu is displayed.

Rows = number
    Defines the number of rows in the menu display.

Columns = number
    Defines the number of columns in the menu display. If neither Rows nor Columns is defined, then the built-in menu heuristic is used for determining the number of rows and columns in the menu.

Multiple
    If this keyword is present, the menu is multi-select. Otherwise, the menu is a single select menu.

Name = item name
    Follows the menu attributes and specifies the name displayed in the menu. This keyword is returned to the caller when this item is selected.

SEE ALSO

menu(3T), form(3T), tam(3T).

NAME

size – print section sizes of common object files

SYNOPSIS

size [**–o**] [**–x**] [**–V**] files

DESCRIPTION

The *size* command produces section size information for each section in the common object files. The size of the text, data, bss (uninitialized data), and shared library sections are printed along with the total size of the object file. If an archive file is input to the *size* command the information for all archive members is displayed.

Numbers will be printed in decimal unless either the **–o** or the **–x** option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **–V** flag will supply the version information on the *size* command.

SEE ALSO

as(1), cc(1), ld(1), a.out(4), ar(4).

DIAGNOSTICS

size: name: cannot open
  if *name* cannot be read.

size: name: bad magic
  if *name* is not an appropriate common object file.

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time as in:

```
(sleep 105; command)&
```

or to execute a command every so often, as in:

```
while true
do
        command
        sleep 37
done
```

SEE ALSO

alarm(2), sleep(3C).

BUGS

Time must be less than 2147483647 seconds.

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	"Dictionary" order: 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 *nroff*(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*, *traveled*, 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* requests), *unless* the names of such included files begin with **/usr/lib**. Under the **−l** option, *spell* will follow the chains of *all* included files.

Under the +*local_file* option, words found in *local_file* are removed from *spell*'s output. *Local_file* is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to *spell*'s own spelling list) for each job.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective with respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine, and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings (see *FILES*). Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g., thier=thy−y+ier) that would otherwise pass.

Three routines help maintain and check the hash lists used by *spell*:

hashmake Reads a list of words from the standard input and writes the corresponding nine-digit hash code on the

	standard output.
spellin	Reads *n* hash codes from the standard input and writes a compressed spelling list on the standard output.
hashcheck	Reads a compressed *spelling_list* and recreates the nine-digit hash codes for all the words in it; it writes these codes on the standard output.

FILES

D_SPELL=/usr/lib/spell/hlist[ab]	hashed spelling lists, American & British
S_SPELL=/usr/lib/spell/hstop	hashed stop list
H_SPELL=/usr/lib/spell/spellhist	history file
/usr/lib/spell/spellprog	program

SEE ALSO

deroff(1), eqn(1), sed(1), sort(1), tbl(1), tee(1).

BUGS

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions; typically, these are kept in a separate local file that is added to the hashed *spelling_list* via ***spellin***.
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 in its stead, then the standard input file is used.

SEE ALSO

bfs(1), csplit(1).

NAME

strip – strip symbol and line number information from a common object file

SYNOPSIS

strip [−l] [−x] [−**r**] [−**s**] [−V] file-names

DESCRIPTION

The *strip* command strips the symbol table and line number information from common object files, including archives. Once this has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

−l	Strip line number information only; do not strip any symbol table information.
−x	Do not strip static or external symbol information.
−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	Version of strip command executing.

If there are any relocation entries in the object file and any symbol table information is to be stripped, *strip* will complain and terminate without stripping *file-name* unless the **−r** flag is used.

If the *strip* command is executed on a common archive file (see *ar*(4)) the archive symbol table will be removed. The archive symbol table must be restored by executing the *ar*(1) command with the **s** option before the archive can be link edited by the *ld*(1) command. *Strip*(1) will instruct 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/strp??????

SEE ALSO

as(1), cc(1), ld(1), ar(4), a.out(4).

DIAGNOSTICS

strip: name: cannot open
if *name* cannot be read.

strip: name: bad magic
if *name* is not an appropriate common object file.

strip: name: relocation entries present; cannot strip
if *name* contains relocation entries and the **−r** flag is not used, the symbol table information cannot be stripped.

NAME

stty – set the options for a terminal

SYNOPSIS

stty [**–a**] [**–g**] [options]

DESCRIPTION

Stty sets certain terminal I/O options for the device that is the current standard input; without arguments, it reports the settings of certain options; 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 may be found in *termio*(7) for asynchronous lines. 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) DATA-PHONE 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**, **ctab**, **min**, or **time** (**ctab** is used with **−stappl;** 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 *UNIX System Administrator's Manual*.

NAME

su – become super-user or another user

SYNOPSIS

su [–] [name [arg . . .]]

DESCRIPTION

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

To use *su*, the appropriate password must be supplied (unless one is already super-user). If the password is correct, *su* will execute 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 super-user 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:/usr/local/bin** for root. Note that the **.profile** can check *arg0* for **–sh** or **–su** to determine how it was invoked.

FILES

/etc/passwd	system's password file
$HOME/.profile	user's profile

SEE ALSO

env(1), login(1M), 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 also prints the number of blocks in the file. It is typically used to look for bad spots, or to validate a file communicated over some transmission line. The option **–r** causes an alternate algorithm to be used in computing the checksum.

SEE ALSO

wc(1).

DIAGNOSTICS

"Read error" is indistinguishable from end of file on most devices; check the block count.

NAME

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 will flush all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point will be saved. See *sync*(2) for details.

SEE ALSO

sync(2).

NAME

tabs – set tabs on a terminal

SYNOPSIS

tabs [tabspec] [**+m**n] [**–T**type]

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., UNIX "standard" tabs. The lowest column number is 1. Note that for *tabs*, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0, e.g., the DASI 300, DASI 300s, and DASI 450.

–*code*	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 UNIX "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 will work for many terminals.

+m*n* The margin argument may be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n+1* the left margin. If +m is given without a value of *n*, the value assumed is 10. For a TermiNet, the first value in the tab list should be 1, or the margin will move even further to the right. The normal (leftmost) margin on most terminals is obtained by **+m0**. The margin for most terminals is reset only when the +m flag is given explicitly.

Tab and margin setting is performed via the standard output.

DIAGNOSTICS

illegal tabs	when arbitrary tabs are ordered incorrectly.
illegal increment	when a zero or missing increment is found in an arbitrary specification.

unknown tab code	when a "canned" code cannot be found.
can't open	if −−*file* option used, and file can't be opened.
file indirection	if −−*file* option used and the specification in that file points to yet another file. Indirection of this form is not permitted.

SEE ALSO

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 is willing to set 40.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [±[number][**lbc**[**f**]]] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning, or −*number* from the end of the input (if *number* is null, the value 10 is assumed). *Number* is counted in units of lines, blocks, or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

With the −**f** ("follow") option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop, wherein it sleeps for a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail -f fred
```

will print the last ten lines of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed. As another example, the command:

```
tail -15cf fred
```

will print the last 15 characters of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed.

SEE ALSO

dd(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 Create a new tape; 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 only be used 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* not to restore the modification times. The modification time of the file will be the time of extraction.

FILES

/dev/mt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if not enough memory is 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 file-name 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* or *troff* (not included on the UNIX PC). 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 re-formatted 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 (**;**).

Next come 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 key-letter, optionally followed by specifiers that determine the font and point size of the corresponding item, that indicate where vertical bars are to appear between columns, that determine column width, inter-column spacing, etc. The available key-letters 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, followed finally by **.TE**. Within such data lines, data items are normally separated by tab characters.

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 file names 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 we let → represent 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

TBL−A Program to Format Tables in the *UNIX System Document Processing Guide*.
cw(1), eqn(1), mm(1), nroff(1), mm(5).

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 on to the next page. In this wait state, the command **e** will *suppress* the screen erase before the next page; **s***n* will cause the next *n* pages to be skipped; and **!***cmd* will send *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).

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 elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The 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

times(2).

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

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
```

The following UNIX PC files are linked to either true or false:

/bin/mc68k
/bin/pdp11l
/bin/u370
/bin/u3b
/bin/vax

SEE ALSO

sh(1).

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

tset – set terminal modes

SYNOPSIS

tset [options] [**–m** [*ident*][*test baudrate*]:*type* ...] [type]

DESCRIPTION

Tset causes terminal dependent processing such as setting erase and kill characters, setting or resetting delays, and the like. It first determines the *type* of terminal involved, names for which are specified by the */etc/termcap* data base, and then does necessary initializations and mode settings. In the case where no argument types are specified, *tset* simply reads the terminal type out of the environment variable *TERM* and re-initializes the terminal. The rest of this manual concerns itself with type initialization, done typically once at login, and options used at initialization time to determine the terminal type and set up terminal modes.

When used in a startup script *.profile* it is desirable to give information about the types of terminal usually used on terminals which are not hardwired. These ports are initially identified as being *dialup* or *plugboard* or *arpanet*, etc. To specify what terminal type is usually used on these ports **–m** is followed by the appropriate port type identifier, an optional baud-rate specification, and the terminal type to be used if the mapping conditions are satisfied. If more than one mapping is specified, the first applicable mapping prevails. A missing type identifier matches all identifiers.

Baud rates are specified as with *stty*(1), and are compared with the speed of the diagnostic output (which is almost always the control terminal). The baud rate test may be any combination of: >, =, <, @, and !; @ is a synonym for = and ! inverts the sense of the test. To avoid problems with metacharacters, it is best to place the entire argument to **–m** within "´" characters.

Thus

```
tset –m ´dialup>300:adm3a´ –m dialup:dw2 –m
´plugboard:?adm3a´
```

causes the terminal type to be set to an *adm3a* if the port in use is a dialup at a speed greater than 300 baud; to a *dw2* if the port is (otherwise) a dialup (i.e. at 300 baud or less). If the *type* above begins with a question mark, the user is asked if s/he really wants that type. A null response means to use that type; otherwise, another type can be entered which will be used instead. Thus, in this case, the user will be queried on a plugboard port as to whether they are using an *adm3a*. For other ports the port type will be taken from the **/etc/ttytype** file or a final, default *type* option may be given on the command line not preceded by a **–m**.

It is often desirable to return the terminal type, as specified by the **–m** options, and information about the terminal to a shell's environment. This can be done using the **–s** option; using the Bourne shell, *sh*(1):

```
eval `tset -s options...`
```

These commands cause *tset* to generate as output a sequence of shell commands which place the variables *TERM* and *TERMCAP* in the environment; see *environ*(5).

Once the terminal type is known, *tset* engages in terminal mode setting. This normally involves sending an initialization sequence to the terminal and setting the single character erase (and optionally the line-kill (full line erase)) characters.

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is the default erase character ('#' on standard systems), the erase character is changed to a Control-H (backspace).

The options are:

−**e**	set the erase character to be the named character *c* on all terminals, the default being the backspace character on the terminal, usually ^H.
−**k**	is similar to −**e** but for the line kill character rather than the erase character; *c* defaults to ^X (for purely historical reasons); ^U is the preferred setting. No kill processing is done if −**k** is not specified.
−**I**	suppresses outputting terminal initialization strings.
−**Q**	suppresses printing the "Erase set to" and "Kill set to" messages.
−~~**S**~~ s (LOWERCASE)	Outputs the strings to be assigned to *TERM* and *TERMCAP* in the environment rather than commands for a shell.

FILES

/etc/ttytype	terminal id to type map database
/etc/termcap	terminal capability database

SEE ALSO

sh(1), stty(1), environ(5), ttytype(5), termcap(5)

BUGS

Should be merged with *stty*(1).

NOTES

For compatibility with earlier versions of *tset* a number of flags are accepted whose use is discouraged:

−**d** type	equivalent to −**m** dialup:type
−**p** type	equivalent to −**m** plugboard:type
−**a** type	equivalent to −**m** arpanet:type
−**E** c	Sets the erase character to *c* only if the terminal can backspace.
−	prints the terminal type on the standard output
−**r**	prints the terminal type on the diagnostic output.

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 path name 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 path name, 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" if the standard input is not a synchronous terminal and **–l** is specified.
"not a tty" if the standard input is not a terminal and **–s** is not specified.

NAME

uahelp – user agent help process

SYNOPSIS

uahelp –h helpfile [**–t** title] [**–d** debugfile]

DESCRIPTION

Uahelp is a help facility which is driven by a text file (*helpfile*). The syntax of this file is described below.

Title, if specified, is the title of the initial help display.

If the **–d** (debug) option is specified, then as *helpfile* is being parsed, the lines are written to *debugfile*. When a syntax error occurs during the parsing of *helpfile*, then *uahelp* displays an error message and quits. The line containing the error is the last line written to *debugfile*. This option is used to debug new *helpfiles*.

Helpfile is an ordinary ASCII text file, with a "keyword = value" syntax. The following keywords are defined:

Keyword	**Value**
Wlabel	Window label
Contents	Lists of help displays in this file
Name	Name of current help display
Llabel	Long screen label for current display
Slabel	Short screen label for current display
Branch	List of help displays available via SLK's from the current help display
Title	Title of current help display
Text	Text of current help display

All keywords must be case correct and followed by an equal sign (=) and a value. The **Wlabel** and **Contents** keywords must be defined at the beginning of the *helpfile*, and they are followed by a series of definitions of the individual help displays, one for each display listed under **Contents**.

The individual help displays begin with a **Name** definition, which must be one of the names listed under **Contents**. The remaining keyword definitions apply to the current help display, up until the **Text** keyword, which terminates the help display definition.

The value of the **Contents** and **Branch** keywords must consist of a list of one or more help display names. These names must be separated by commas, and the final one must be terminated with a new line character. The value of the **Name** keyword is a single help display name, 16 characters or less. The value of the **Wlabel**, **Llabel**, **Slabel**, and **Title** keywords are strings enclosed in double quotes (" ").

The value of the **Text** keyword is text in ADF format (see *ADF*(4)). The following embedded codes are recognized:

\CEN\	Center the line
\IND\	Indent to the next tab stop
\UL\	Begin underlining

\US\	End underlining
\BL\	Begin bold text (reverse video)
\BS\	End bold text
\EOT\	End of text

EXAMPLE

The following is the beginning of a help file, which might be used for a word processing help facility. It is recommended that all help files include a help display called "Using help," as in this example.

```
Wlabel = "Word processor help"
Contents = Using help, Getting started, Cursor,
Insert, Edit, Format
Name = Using help
LLlabel = " USING    HELP"
Slabel = "  HELP"
Branch = Using help, Getting started
Title = "How to use the HELP facility"
Text = You can use the HELP facility in two different ways:

Normal help displays consist of a description which \
displayed in a window. If the description doesn't fit \
in the window, the ROLL UP and ROLL DOWN keys may be \
used to view the rest of the display. The screen \
labeled keys at the bottom of the display contain the \
names of other help displays. Press one of these function \
keys to view a different help display.

Press function key F1 (labeled\UL\TABLE OF
CONTENTS\US\ on the screen) \
to see a listing of all available help \
displays. Select the help display you want with the \
cursor and press ENTER.

In either case, pressing EXIT ends the help display.\EOT\

Name = Getting started
Llabel = "GETTING STARTED"
Slabel = "STARTING"
Branch = Using help, Cursor, Insert, Edit, Format
Title = "Starting to use the word processor"
Text =

                    .
                    .
                    .
```

Note that the returns are all escaped with the backslash (\), except for the hard returns at the end of paragraphs.

SEE ALSO

message(3T), ADF(4).

CAVEATS

Uahelp arbitrarily limits help files to 100 distinct displays, and each display is limited to 100 lines.

NAME

uaupd – update user agent special files

SYNOPSIS

uaupd -r ObjectName [**-a** UpdateFile] filename

DESCRIPTION

Uaupd updates the special file named in the command line. This file is assumed to reside in the directory **/usr/lib/ua**.

The **-r** option must be specified, and removes the entry associated with the given *ObjectName* from the special file.

The **-a** option adds the contents of the *UpdateFile* to the special file. The format of the user agent special files is described in *ua*(4).

SEE ALSO

ua(4).

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**). Umask 022 is the default on the UNIX PC.

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

umodem – remote file transfer program for CP/M terminals

SYNOPSIS

umodem – [**rb** | **rt** | **sb** | **st**] [**q**] [**l**] [**m**] [**d**] [**y**] [**7**] *filename*

DESCRIPTION

Umodem cooperates with the MODEM.COM, YAM.COM, or similar program, running on a CP/M-based intelligent terminal, to perform a file transfer. The integrity of the transfer is enhanced by use of a block checksum for error detection, and block retransmission for error correction.

Umodem requires exactly one of the following commands:

rb Receive Binary–transfer a file *from* the terminal, in raw binary mode. Every byte of the file will be transferred intact. This mode is usually used to transfer, for example, .COM files.

rt Receive Text–transfer a file *from* the terminal, in text mode. In this mode the program attempts to convert from the CP/M text file format to the UNIX format on-the-fly, by stripping carriage-return characters, and by ceasing to store data after a control-Z is detected.

sb Send Binary–transfer a file *to* the terminal, in raw binary mode. Every byte of the file will be transferred intact. This mode is usually used to transfer, for example, .COM files.

st Send Text–transfer a file *to* the terminal, in text mode. In this mode the program attempts to convert from the UNIX text file format to the CP/M format on-the-fly, by adding carriage-return characters, and by appending a control-Z to the end of the file.

In addition, *umodem* recognizes the following options:

q Quiet option–the initial "boiler plate" of program name, file size, etc., is suppressed.

l Logfile option–enables logging the progress of the file transfer. This option is primarily useful for debugging.

m "Mung-mode" option–unless this option is specified, an attempt to receive a *filename* that already exists will be denied. With this option, the existing file is overwritten.

d Delete the logfile, if it exists, before starting.

y Display file status (size) information only.

7 Seven-bit transfer option–strip off the high-order bit of each byte before it is sent (–**st** case) or stored (–**rt** case). This option is valid only for text-mode transfers.

EXAMPLES

To transfer MODEM.COM (an executable binary file) to UNIX:

```
umodem -rb modem.com
```

To transfer MYDOC.TXT (a WordStar™ text file) to UNIX, and get rid of the high–order formatting bits that WordStar™ loves to embed in the file:

```
umodem -rt7 mydoc.txt
```

To transfer **foo.c** (a UNIX C–source file) to the CP/M terminal:

```
umodem -st foo.c
```

FILES

$HOME/umodem.log	created or appended to if the –**l** option is specified.

SEE ALSO

MODMPROT.001–Ward Christensen's description of the MODEM protocol

MODEM7xx.DOC–Documentation for the MODEM7 series of CP/M smart terminal programs, written in 8080 assembly language

YAMDOC.RNO–Documentation for the YAM smart terminal program, written in BDS C.

BUGS

The program supports only the checksum block error check, and not the more robust CRC.

The program supports neither the MODEM7 nor the YAM batch file transfer protocols. Only single file transfers are supported.

NAME

uname – print name of current UNIX system

SYNOPSIS

uname [**–snrvma**]

DESCRIPTION

Uname prints the current system name of UNIX 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 comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. *Input* and *output* should always be different. Note that repeated lines must be adjacent in order to be found; see *sort*(1). If the **–u** flag is used, just the lines that are not repeated in the original file are output. The **–d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **–u** and **–d** mode outputs.

The **–c** option supersedes **–u** and **–d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

–*n* The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+*n* The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

comm(1), sort(1).

NAME

units – conversion program

SYNOPSIS

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have:  inch
You want:  cm
        * 2.540000e+00
        / 3.937008e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have:  15 lbs force/in2
You want:  atm
        * 1.020689e+00
        / 9.797299e-01
```

Units only does multiplicative scale changes; thus it can convert Kelvin to Rankine, but not Celsius to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter,
c	speed of light,
e	charge on an electron,
g	acceleration of gravity,
force	same as **g**,
mole	Avogadro's number,
water	pressure head per unit height of water,
au	astronomical unit.

Pound is not recognized as a unit of mass; **lb** is. Compound names are run together, (e.g. **lightyear**). British units that differ from their U.S. counterparts are prefixed thus: **brgallon**. For a complete list of units, type:

```
cat /usr/lib/unittab
```

FILES

/usr/lib/unittab

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 file name may be a path name 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 forward information.

The shell metacharacters **?**, ***** and **[...]** appearing in *path-name* will be expanded on the appropriate system.

Path names may be one of:

(1) a full path name;

(2) a path name preceded by *~user* where *user* is a login name on the specified system and is replaced by that user's login directory;

(3) a path name 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 path name for the remote system the copy will fail. If the *destination-file* is a directory, the last part of the *source-file* name is used.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*:

–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 will only be 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 will very likely not be able to fetch files by path name; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary path names. 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* will be owned by *uucp*.
The −**m** option will only work sending files or receiving a single file. Receiving multiple files specified by special shell characters **? * [...]** will not activate the −**m** option.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

uustat [options]

DESCRIPTION

Uustat will display the status of, or cancel, previously specified *uucp* commands, or provide 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 super-user.
–**r***jobn*	Rejuvenate *jobn*. That is, *jobn* is touched so that its modification time is set to the current time. This prevents *uuclean* from deleting the job until the job's modification time reaches the limit imposed by *uuclean.*
–**c***hour*	Remove the status entries which are older than *hour* hours. This administrative option can only be initiated by the user **uucp** or the super-user.
–**u***user*	Report the status of all *uucp* requests issued by *user*.
–**s***sys*	Report the status of all *uucp* requests which 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* are 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
```

will print the status of all *uucp* requests that were issued by user *hdc* to communicate with system *mhtsa* within the last 72 hours. The meanings of the job request status are:

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 meanings of the machine accessibility status are:

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-to-UNIX 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 path name on your machine. Destination has the form:

system!*user*

where *system* is taken from a list of system names that *uucp* knows about (see *uuname*). *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 will only search 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 will gather zero or more files from various systems, execute a command on a specified system and then send standard output to a file on a specified system. Note that, for security reasons, many installations will limit the list of commands executable on behalf of an incoming request from *uux*. Many sites will 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 and file names may be prefixed by *system-name*!. A null *system-name* is interpreted as the local system.

File names may be one of

(1) a full path name;

(2) a path name preceded by ~*xxx* where *xxx* is a login name on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

As an example, the command

```
uux "!diff usg!/usr/dan/f1 pwba!/a4/dan/f1 > !f1.diff"
```

will get the **f1** files from the "usg" and "pwba" machines, execute a *diff* command and put 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 will attempt to get all files to the execution system. For files which are output files, the file name must be escaped using parentheses. For example, the command

```
uux a!uucp b!/usr/file \(c!/usr/file\)
```

will send a *uucp* command to system "a" to get **/usr/file** from system "b" and send it to system "c".

Uux will notify you if the requested command on the remote system was 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/lib/uucp/spool	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., **r**1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc. which may exist) or invalid (e. g., **r**1.0 or **r**1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.
–**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 will produce a **core** dump.

NAME

vc – version control

SYNOPSIS

vc [**–a**] [**–t**] [**–c**char] [**–s**] [keyword=value ... keyword=value]

DESCRIPTION

The *vc* command copies lines from the standard input to the standard output under control of its *arguments* and *control statements* encountered in the standard input. In the process of performing the copy operation, user declared *keywords* may be replaced by their string *value* when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests (in control statements) of keyword values specified in control statements or as *vc* command arguments.

A control statement is a single line beginning with a control character, except as modified by the **–t** keyletter (see below). The default control character is colon (**:**), except as modified by the **–c** keyletter (see below). Input lines beginning with a backslash (\) followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or less alphanumerics; the first must be alphabetic. A value is any ASCII string that can be created with *ed*(1); a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Replacement of keywords by values is done whenever a keyword surrounded by control characters is encountered on a version control statement. The **–a** keyletter (see below) forces replacement of keywords in *all* lines of text. An uninterpreted control character may be included in a value by preceding it with \. If a literal \ is desired, then it too must be preceded by \.

Keyletter Arguments

–a Forces replacement of keywords surrounded by control characters with their assigned value in *all* text lines and not just in *vc* statements.

–t All characters from the beginning of a line up to and including the first *tab* character are ignored for the purpose of detecting a control statement. If one is found, all characters up to and including the *tab* are discarded.

–c*char* Specifies a control character to be used in place of **:**.

–s Silences warning messages (not error) that are normally printed on the diagnostic output.

Version Control Statements

:dcl keyword[, **...**, keyword]

Used to declare keywords. All keywords must be declared.

:asg keyword=value

Used to assign values to keywords. An **asg** statement overrides the assignment for the corresponding keyword on the *vc* command line and all previous **asg**'s for that keyword. Keywords declared, but not assigned values have null values.

:if condition
 .
 .
 .
:end

Used to skip lines of the standard input. If the condition is true all lines between the *if* statement and the matching *end* statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control statements. Note that intervening *if* statements and matching *end* statements are recognized solely for the purpose of maintaining the proper *if-end* matching.

The syntax of a condition is:

```
<cond>   ::= [ "not" ] <or>
<or>     ::= <and> | <and> "|" <or>
<and>    ::= <exp> | <exp> "&" <and>
<exp>    ::=  "("  <or>  ")"  |  <value>  <op>
<value>
<op>     ::= "=" | "!=" | "<" | ">"
<value>  ::=  <arbitrary ASCII string>  |  <numeric
string>
```

The available operators and their meanings are:

=	equal
!=	not equal
&	and
|	or
>	greater than
<	less than
( )	used for logical groupings
not	may only occur immediately after the *if*, and when present, inverts the value of the entire condition

The > and < operate only on unsigned integer values (e. g.: 012 > 12 is false). All other operators take strings as arguments (e. g.: 012 != 12 is true). The precedence of the operators (from highest to lowest) is:

```
= != > <     all of equal precedence
&
|
```

Parentheses may be used to alter the order of precedence. Values must be separated from operators or parentheses by at least one blank or tab.

::text
: Used for keyword replacement on lines that are copied to the standard output. The two leading control characters are removed, 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
: Prints the given message on the diagnostic output.

:err message
: Prints the given message followed by:

  ERROR: err statement on line **...** (915)

  on the diagnostic output. *Vc* halts execution, and returns an exit code of 1.

DIAGNOSTICS

Use *help*(1) for explanations.

EXIT CODES

0 – normal
1 – any error

NAME

vi, view – screen oriented (visual) display editor based on ex

SYNOPSIS

vi [**–t** tag] [**–r**] [**+***command*] [**–l**] [**–w***n*] **–x** name ...

DESCRIPTION

Vi (visual) is a display oriented text editor based on *ex*(1) *View* is synonymous with *vi*. *Ex* and *vi* run the same code; it is possible to get to the command mode of *ex* from within *vi* and vice-versa.

Note that the ability to edit encrypted files is present only in the domestic (U.S.) version of the UNIX PC software.

COMMANDS

The following summarizes the *vi* commands and procedures. The *Introduction to Display Editing with Vi* provides full details on using *vi*.

NOTATION AND SPECIAL KEYS

^	Denotes the CONTROL key (Ctrl on the UNIX PC) to be held down while the following character is typed.
↑	Used to show the caret (^) should be typed.
[n]	Optional number of repetitions preceding a command. Do not type [ ]. In most cases omitting *n* defaults to one.
object	The text object—(character, word, sentence, paragraph, or line) that a command operates on.
:	A prefix to a set of commands for file and option manipulation and escapes to the shell. The **:** and later keystrokes appear at the bottom of the screen. The command is terminated with a **<CR>** or **<ESC>**.
<ESC>	ESCAPE key (Esc on the UNIX PC) used to return to command mode. Type **<ESC>** when you are not sure of the current mode. Causes a beep if already in command mode (harmless).
<CR>	Carriage RETURN key.
BS	BACKSPACE key. **^H** on terminals without a backspace key.
DELETE	Sometimes labeled **DEL**, **BREAK**, or **RUBOUT** (shift of the Esc key on the UNIX PC). This key generates an interrupt that tells the editor to stop what it is doing.

ENTERING THE VI EDITOR

Note: Follow entry with <CR>.

vi *file*	Edit at first line of *file*
vi	Edit a new empty *file*
vi + *n file*	Edit at *n* line in *file*
vi + *file*	Edit at last line in *file*
vi −r	List saved files
vi −r *file*	Recover *file* and edit saved *file*
vi *file1, file2, ...*	Edit *file1*; *file2*; ... (after editing *file1* enter **:n** for each remaining file)
vi −t *tag*	Edit at *tag* file in *tags* file
vi +/*pat file*	Search for and edit at *pattern* in *file*
view *file*	Read only view of file

LEAVING VI EDITOR

:q<CR>	Quit *vi* when no changes have occurred since last write
:q!<CR>	Quit *vi*, do not save changes since last write
:wq<CR>	Write and quit (exit *vi*, saving changes)
ZZ	Write and quit (exit *vi*, saving changes)

POSITIONING THE CURSOR

File Positioning

[n]^**F**	Forward *[n]* full screens
[n]^**B**	Backward screens
[n]^**D**	Scroll down (default is half screen)
[n]^**U**	Scroll up (default is half screen)
[n]^**E**	Scroll down 1 line
[n]^**Y**	Scroll up 1 line
*[n]***G**	Go to line *n* (default is last line of file)
[n]//pat	Go to next line matching *pat*
[n]?pat	Previous line matching *pat*
*[n]***n**	Repeat last / or ?
*[n]***N**	Reverse last / or ?
[n]//pat/+m	*m*th line after *pat*
[n]?pat?−m	*m*th line before *pat*

Screen Positioning

*[n]***H**	To *n*th line from top of display. Without *n*, to top
*[n]***L**	To *n*th line from bottom of display. Without *n*, to bottom
M	To middle line of display

Line Positioning

0	Beginning of line

[n]$	End of line
[n]+	Next line, at first non-white
[n]–	Previous line, at first non-white
[n]<CR>	Return, same as +
[n]↓| or **j**	Next line, same column
[n]↑| or **k**	Previous line, same column

Character Positioning Within a Line

[n]↑	First non-white
*[n]***h** or →	Forward one character
*[n]***l** or ←	Backward one character
*[n]*spacebar	Same as →
*[n]*backspace	Backwards one character
*[n]***^H**	Same as ← or backspace
*[n]***f***x*	Find *x* forward
*[n]***F***x*	Find *x* backward
*[n]***t***x*	Move up to *x* forward
*[n]***T***x*	Move up to *x* backward
*[n]***;**	Repeat last **f**, **F**, **t**, or **T**
*[n]***,**	Inverse of **;**
[n]|	Move to specified column number *n*

Word Positioning

*[n]***w**	Move forward to beginning of word. Punctuation and strings of punctuation count as words.
*[n]***b**	Move back to beginning of word. Punctuation and strings of punctuation count as words.
*[n]***e**	Move forward to end of word. Punctuation and strings of punctuation count as words.
*[n]***W**	Move forward to beginning of word. Punctuation ignored.
*[n]***B**	Move back to beginning of word. Punctuation ignored.
*[n]***E**	Move forward to end of word. Punctuation ignored.

Sentence, Paragraph, Heading Positioning

*[n]***)**	Forward to next sentence
*[n]***(**	Back a sentence
*[n]***}**	Forward to next paragraph
*[n]***{**	Back a paragraph

[n]]]	Forward to next heading
[n] [[	Back a heading

CREATING TEXT

a*text*<ESC>	Append after cursor, until <ESC>
i*text*<ESC>	Insert before cursor
A*text*<ESC>	Append at end of line
I*text*<ESC>	Insert before first non-blank
o*text*<ESC>	Open line below
O*text*<ESC>	Open above

MAKING CORRECTIONS DURING TEXT CREATION

^W	Erase last word during an insert
kill	Kill the insert on this line (usually @, **^X**, or **^U**)
*[n]*BS	Erase last character
^H	Erase last character
\	Escapes **^H**, your erase and kill
<ESC>	Ends insertion, back to command mode
^?	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

MODIFYING TEXT

Changing Text

~	Switch character from lowercase to uppercase and vice versa
*[n]***C***text*<ESC>	Change from cursor to end of line (same as **c$**)
*[n]***R***text*<ESC>	Replace characters
*[n]***S***text*<ESC>	Substitute on lines
*[n]*cobj*text*<ESC>	Change the specified object (word) to the following *text*
*[n]***r***x*	Replace character with *x*
*[n]***s***text*<ESC>	Replace a character with a *text* string
*[n]***cc***text*<ESC>	Change a whole line

Deleting Text

D	Delete from cursor to end of line
*[n]***x**	Delete a character
*[n]***X**	Delete character to left of cursor
*[n]*d*(object)*	Delete the specified *object* (word, sentence, paragraph, etc.)

*[n]*dd	Delete a line
Moving Text	
"*r*	Named register *r* that save delete commands. Legal values of *r* are letters **a** through **z**.
"*r***p**	Puts deleted text from registers "*r* after or below cursor
"*r***P**	Puts deleted text from registers "*r* before or above cursor
p	Puts last deleted text after or below cursor
P	Puts last deleted text before or above cursor
Copying Text	
"*r*	Named register *r* that can precede a yank command. Legal values of *r* are letters **a** through **z**.
y*[n]*object	Yanks a copy of the following object into a register
*[n]***Y**	Yanks a copy of the current line into a register
*[n]***yy**	Same as **Y**
"*r***p**	Puts yanked text from register "*r* after or below cursor
"*r***P**	Puts yanked text from register "*r* before or above cursor
p	Puts last yanked text after or below cursor
P	Puts last yanked text before or above cursor

UNDOING, REDOING, RETRIEVING

u	Undo last change
U	Restore current line
.	Repeat last change
"*h***p**	Retrieve one of last 9 deletes; *h* is a hidden register numbered 1 through 9. Retrieved in reverse order.

DOING GLOBAL SEARCHES AND CHANGES

Note: Follow entry with <CR>.

:g/*text*	Move cursor to last line in file with *text*
:g/*text***/p**	Print all lines with *text*
:g/*text***/nu**	Print all lines and line numbers with *text*
:*[m]***,***[n]***g/***text*	Move cursor to *n* line in file with *text*
:*[m]***,***[n]***g/***text***/p**	Print all lines with *text* from line *m* to *n*
:*[m]***,***[n]***g/***text***/nu**	Print all lines and line numbers with *text* from line *m* to *n*
:g/*text***/s//***newtext*	Change first appearance of *text* in each line in

	file to *newtext*
:g/*text***/s//***newtext***/p**	Change first appearance of *text* in each line in file to *newtext* and print each changed line
:g/*text***/s//***newtext***/c**	List one at a time each line with *text* and change as required to *newtext* using a **y**<CR>
:*[m]***,***[n]***g/***text***/s//***newtext*	Change first appearance of *text* in each line in file to *newtext*
:*[m]***,***[n]***g/***text***/s//***newtext***/p**	Change first appearance of *text* in lines from *m* to *n* to *newtext* and print each changed line
:*[m]***,***[n]***g/***text***/s//***newtext***/c**	List one at a time each line with *text* from *m* to *n* and change as required to *newtext* using a **y**<CR>

MANIPULATING FILES

Copy From Another File

:r *file*<CR>	Copy *file* into buffer after current line
:*[n]***r** *file*<CR>	Copy *file* to buffer after *n*th line

Copy To Another File

Note: Follow entry with <CR>.

:w *file*	Write the current file to *file*
:w! *file*	Overwrite existing *file* with *file*
:w>>*file*	Add current file to end of *file*
:*[m]***,***[n]***w** *file*	Write lines *m* through *n* to *file*
:*[m]***,***[n]***w!** *file*	Overwrite existing *file* with *file* containing lines *m* through *n*
:*[m]***,***[n]***w>>***file*	Add lines *m* through *n* to end of *file*

Edit Current File

:w<CR>	Write changes to current file
:w *file*<CR>	Write *file* to current unnamed file
"**e!**<CR>	Reedit current file, discarding changes since last write
:f<CR>	Show current file and line
^G	Synonym for **:f**
:ta *tag*<CR>	To tag file entry *tag*
^]	**:ta**, following word is *tag*

Edit Other Files From Current File

:e *file*<CR>	Edit *file* when write has occurred in current file, return to shell after edit, changes not lost in current file

:e! *file*<CR>	Edit *file* when no write has occurred in current file, return to shell after edit, changes list in current file
:e + *name*<CR>	Edit starting at end
:e + *n*<CR>	Edit starting at line *n*
:n<CR>	Edit next file in list when *vi* was called with more than one file
:n *args*<CR>	Specify new list of files to be edited
:e #<CR>	Edit alternate file when two files are being edited
^↑	Synonym for **:e** #.

ESCAPING TO THE SHELL

:sh<CR>	Start a separate shell (to run several commands), return with ^D
:!*command*<CR>	Run one shell *command*, then return to current buffer

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

MISCELLANEOUS OPERATIONS

.	Repeat the last append, insert, open, delete, change, or put command
~	Switch character from lowercase to uppercase and vice versa
^?	Delete or rubout interrupts
i<CR><ESC>	Split a line before the cursor
a<CR><ESC>	Split a line after the cursor
^L	Reprint screen if **^?** scrambles it
J	Join lines
:nu<CR>	Line number cursor is on
xp	Switch characters

SETTING OPTIONS
Initializing Options

:set *x*<CR>	Enable option *x*
:set no*x*<CR>	Disable option *x*
:set *x*=*val*<CR>	Assign a value to *x* option
:set<CR>	Show changed options
:set all<CR>	Show all options

:set *x*?<CR>   Show value of option *x*

Options

autoindent, ai (default: noai)
When on, in the append, change, insert, open, or substitute mode a new line will be started at same indent as previous line.

audoprint, ap (default: ap)
Causes the current line to be printed after each delete, copy, join, move, substitute, t, undo or shift command. This has the same effect as supplying a trailing **p** to each such command. The *autoprint* is suppressed in globals and only applies to the last of many commands on a line.

autowrite, aw (default: noaw)
Causes the contents of the buffer to be written to the current file (if you have modified it) and gives a next, rewind, tab, or ! command, or a ^↑ (switch files) or ^] (tag goto) command. Note: the command does not autowrite. In each case, there is an equivalent way of switching when the *autowrite* option is set to avoid the autowrite (ex for next, rewind! for rewind, tag! for tag, shell for !, and :e #nd for a :ta! command).

beautify, bf (default: nobeautify)
Causes all control characters except tab, newline, and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. The *beautify* option does not apply to command input.

directory, dir (default: dir=/tmp)
Specifies the directory in which *vi* places its buffer file. If this directory is not writable, then the editor will exit abruptly when it fails to be able to create its buffer there.

edcompatible (default: noedcompatible)
Causes the presence or absence of **g** and **c** suffixes on substitute commands to be remembered and to be toggled by repeating the suffixes. The suffix **r** makes the substitution be as in the ~ command, instead of line &.

errorbells, eb (default: noeb)
Error messages are preceded by a bell. Bell ringing in *open* and *visual* mode on errors is not suppressed by setting *noeb*. If possible the editor always places the error message in a standout mode of the terminal (such as inverse video) instead of ringing the bell.

hardtabs, ht (default: ht=8)
Gives the boundaries on which terminal hardware tabs are set (or on which the system expands tabs).

ignorecase, ic (default: noic)
All uppercase characters in the text are mapped to lower case in regular expression matching and vice versa, except in character class specifications.

lisp (default: nolisp)
: The *autoindent* option indents appropriately for *lisp code, and the* ( ), { }, [[, and]] commands in *open* and *visual* modes are modified to have meaning for *lisp*.

list (default: nolist)
: All printed lines will be displayed more unambiguously, showing tabs and end-of-lines as in the *list* command.

magic (default: magic for *vi*)
: If *nomagic* is set, the number of regular expression metacharacters is greatly reduced, with only ↑ and $ having special effects. In addition, the metacharacters ˜ and & of the replacement pattern are treated as normal characters. All the normal metacharacters may be made *magic* when *nomagic* is set by preceding them with a \.

mesg (default: mesg)
: Causes write permission to be turned off to the terminal while you are in *visual* mode if *nomesg* is set.

number, nu (default: nonumber)
: Causes all output lines to be printed with line numbers. In addition, each input line will be prompted for by supplying the line number it will have.

open (default: open)
: If *noopen*, the commands *open* and *visual* are not permitted.

optimize opt (default: optimize)
: Throughput of text is expedited by setting the terminal not to do automatic carriage returns when printing more than one (logical) line of output, greatly speeding output on terminals without addressable cursors when text with leading white space is printed.

paragraphs, para (default: para=IPLPPPQPPLIbp)
: Specifies the paragraphs for the { and } operations in *open* and *visual* mode. The pairs of characters in the option's value are the names of the macros which start paragraphs.

prompt (default: prompt)
: Command mode input is prompted for with a colon (**:**).

readonly (default: noreadonly)
: Set by *chmod* shell command to allow read but no write.

redraw (default: noredraw)
: The editor simulates (using great amounts of output) an intelligent terminal on a dumb terminal (e.g., during insertions in *visual* mode the characters to the right of the cursor position are refreshed as each input character is typed). This option is useful only at very high speed.

remap (default: remap)
: If on, macros are repeatedly tried until they are unchanged. For example, if **o** is mapped to **O**, and **O** is

mapped to **I**, then if *remap* is set, **o** will map to *I*; but of *noremap* is set, if will map to **O**.

report (default: report=5)
Specifies a threshold for feedback from commands. Any command which modifies more than the specified number of lines will provide feedback as to the scope of its changes. For commands such as global, *open*, undo, and *visual*, which have potentially more far-reaching scope, the net change in the number of lines in the buffer is presented at the end of the command subject to this same threshold. Thus, notification is suppressed during a global command on the individual commands performed.

scroll (default: scroll=½ window)
Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in *command* mode and the number of lines printed by a *command* mode **z** command (double the value of *scroll*).

sections (default: sections=SHNHH HU)
Specifies the section macros for the [[and]] operations in *open* and *visual* modes. The pairs of characters in the option's value are the names of the macros which start paragraphs.

shell, sh (default: sh=/bin/sh)
Gives the pathname of the shell forked for the shell escape command **!**, and by the *shell* command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw (default: sw=8)
Gives the width a software tabstop used in reverse tabbing with **^D** when using *autoindent* to append text, and by the shift commands.

showmatch, sm (default: nosm)
In *open* and *visual* modes, when a) or } is typed, the cursor moves to the matching (or { for one second if this matching character is on the screen. Extremely useful with *lisp*.

slowopen, slow (default: terminal dependent)
Affects the display algorithm used in *visual* mode, holding off display updating during input of new text to improve throughput when the terminal in use is both slow and unintelligent.

tabstop, ts (default: ts=8)
The editor expands tabs in the input file to be on *tabstop* boundaries for the purposes of display.

taglength, tl (default: tl=0)
Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags (default: tags=tags /usr/lib/tags)
A path of files to be used as tag files for the *tag* command.

A requested tag is searched for in the specified files, sequentially. By default, files called *tags* are searched for in the current directory and in */usr/lib* (a master file for the entire system).

term (default from environment $TERM)
The terminal type of the output device.

terse (default: noterse)
Shorter error diagnostics are produced for the experienced user.

ttytype=
Terminal type defined to system for visual mode. Can be defined before entering visual editor by TERM=type.

warn (default: warn)
Warns if there has been "[No write since last change]" before a **!** command escape.

window (default: window=speed dependent)
The number of lines in a text window in the *visual* command. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600
These are not true options but set *window* only if the speed is slow (300), medium (1200), or high (9600), respectively. They are suitable for an EXINIT and make it easy to change the 8/16/full screen rule.

wrapscan, ws (default: ws)
Searches that use regular expressions in addressing will wrap around past the end of the file.

wrapmargin, wm (default: wm=0)
Defines a margin for automatic wrapover of text during input in *open* and *visual* modes.

writeany, wa (default: nowa)
Inhibit checks normally made before write commands, allowing a write to any file which the system protection mechanism will allow.

FILES

See *ex*(1).

SEE ALSO

ex(1), edit (1), "An Introduction to Display Editing with Vi".

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.*

Moving the cursor backward a screen at a time does not work correctly.

The *[n]* precursor does not work for these commands: **B**, **U**, */pat*, *?pat*, */pat*, */pat/+m*, *?pat?-m*.

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

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus can't be waited for.

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 will be 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
```

will print

```
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

who – who is on the system

SYNOPSIS

who [**–uTlpdbrtas**] [file]

who am i

DESCRIPTION

Who can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process-ID of the command interpreter (shell) for each current UNIX user. It examines the **/etc/utmp** file to obtain its information. If *file* is given, that file is examined. Usually, *file* will be **/etc/wtmp**, which contains a history of all the logins since the file was last created.

Who with the **am i** 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* can list logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the *init* process. These options are:

–u This option lists only those users who are currently logged in. The *name* is the user's login name. The *line* is the name of the line as found in the directory **/dev**. The *time* is the time that the user logged in. The *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 can contain information about where the terminal is located, the telephone number of the dataset, type of terminal if hard-wired, etc.

–T This option is the same as the **–u** option, except that the *state* of the terminal line is printed. The *state* describes whether someone else can write to that terminal. A + appears if the terminal is writable by anyone; a – appears if it is not. **Root** can write to all lines having a + or a – in the *state* field. If a bad line is encountered, a **?** is printed.

–l This option lists only those lines on which the system is waiting for someone to login. The *name* field is **LOGIN** in such cases. Other fields are the same as for user entries except that the *state* field doesn't exist.

–p This option lists any other process which is currently active and has been previously spawned by *init*. The *name* field is the name of the program executed by *init* as found in **/etc/inittab**. The *state*, *line*, and *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 This option displays all processes that have expired and not been respawned by *init*. The *exit* field appears for dead processes and contains the termination and exit values (as returned by *wait*(2)), of the dead process. This can be useful in determining why a process terminated.

−b This option indicates the time and date of the last reboot.

−r This option indicates the current *run-level* of the *init* process.

−t This option indicates the last change to the system clock (via the *date*(1) command) by **root**. See *su*(1).

−a This option processes **/etc/utmp** or the named *file* with all options turned on.

−s This option is the default and lists only the *name*, *line* and *time* fields.

FILES

/etc/utmp
/etc/wtmp
/etc/inittab

SEE ALSO

init(1M) in the *UNIX System Administrator's Manual*.
date(1), login(1M), mesg(1), su(1), wait(2), inittab(4), utmp(4).

NAME

write – write to another user

SYNOPSIS

write user [line]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message:

Message from *yourname* **(tty**??**)** [*date*]**. . .**

to the person you want to talk to. When it has successfully completed the connection it also sends two bells to your own terminal to indicate that what you are typing is being sent.

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal 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 to send to (e.g., **tty00**); otherwise, the first instance of the user found in **/etc/utmp** is assumed and the following message posted:

> *user* is logged on more than one place.
> You are connected to "*terminal*".
> Other locations are:
> *terminal*

Permission to write may be denied or granted by use of the *mesg*(1) command. Writing to others is normally allowed by default. Certain commands, in particular *nroff*(1) and *pr*(1) disallow messages in order to prevent interference with their output. However, if the user has super-user 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*" if the person you are trying to *write* to is not logged in.

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [flags] [command [initial-arguments]]

DESCRIPTION

Xargs combines the fixed *initial-arguments* with arguments read from standard input to execute the specified *command* one or more times. The number of arguments read for each *command* invocation and the manner in which they are combined are determined by the flags specified.

Command, which may be a shell file, is searched for, using one's **$PATH**. If *command* is omitted, **/bin/echo** is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or 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 of quoted strings a backslash (\) will escape the next character.

Each argument list is constructed starting with the *initial-arguments,* followed by some number of arguments read from standard input (Exception: see **–i** flag). Flags **–i**, **–l**, and **–n** determine how arguments are selected for each command invocation. When none of these flags are coded, the *initial-arguments* are followed by arguments read continuously from standard input until an internal buffer is full, and then *command* is executed with the accumulated args. This process is repeated until there are no more args. When there are flag conflicts (e.g., **–l** vs. **–n**), the last flag has precedence. *Flag* values are:

–l*number* *Command* is executed for each non-empty *number* lines of arguments from standard input. The last invocation of *command* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first 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 will be used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* arguments remaining. If option **−x** is also coded, each *number* arguments must fit in the *size* limitation, else *xargs* terminates execution.

−t Trace mode: the *command* and each constructed argument list are echoed to file descriptor 2 just prior to their execution.

−p Prompt mode: the user is asked whether to execute *command* each invocation. Trace mode (**−t**) is turned on to print the command instance to be executed, followed by a **?...** prompt. A reply of **y** (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of *command*.

−x Causes *xargs* to terminate if any argument list would be greater than *size* characters; **−x** is forced by the options **−i** and **−l**. When neither of the options **−i**, **−l**, or **−n** are coded, the total length of all arguments must be within the *size* limit.

−s*size* The maximum total size of each argument list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If **−s** is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each argument and the count of characters in the command name.

−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 will terminate if it receives a return code of **−1** from, or if it cannot execute, *command*. When *command* is a shell program, it should explicitly *exit* (see *sh*(1)) with an appropriate value to avoid accidentally returning with **−1**.

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs -i -t mv $1/{ } $2/{ }
```

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the end of file *log*:

```
(logname; date; echo $0 $*) | xargs >>log
```

The user is asked which files in the current directory are to be archived and archives them into *arch* (1.) one at a time, or (2.) many at a time.

```
1.  ls | xargs -p -l ar r arch
2.  ls | xargs -p -l | xargs ar r arch
```

The following will execute *diff*(1) with successive pairs of arguments originally typed as shell arguments:

```
echo $* | xargs -n2 diff
```

DIAGNOSTICS

Self explanatory.

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 an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, **y.tab.c**, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *lex*(1) is useful for creating lexical analyzers usable by *yacc*.

If the **–v** flag is given, the file **y.output** is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **–d** flag is used, the file **y.tab.h** is generated with the **#define** statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

If the **–l** flag is given, the code produced in **y.tab.c** will *not* contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

Runtime debugging code is always generated in **y.tab.c** under conditional compilation control. By default, this code is not included when **y.tab.c** is compiled. However, when *yacc*'s **–t** option is used, this debugging code will be compiled by default. Independent of whether the **–t** option was used, the runtime debugging code is under the control of **YYDEBUG**, a preprocessor symbol. If **YYDEBUG** has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp,	
yacc.debug, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1).
YACC–Yet Another Compiler Compiler in the *UNIX System 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 file names 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 of 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 of 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 super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

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 beyond the limits of the device. It 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* 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*, was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
: A *fork*, 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*, *brk*, or *sbrk*, a program asks 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*.

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*.

16 EBUSY Mount device busy
: An attempt 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). It will also occur 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*.

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, for example in a path prefix or as an argument to *chdir*(2).

21 EISDIR Is a directory
An attempt 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*, or *kill*; reading or writing a file for which *lseek* 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 no more *opens* can be accepted.

24 EMFILE Too many open files
No process may have more than 80 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading). Also 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 (2,147,483,647 bytes) or ULIMIT; see *ulimit*(2).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek* 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 to make more than the maximum number of links (1000) to a file.

32 EPIPE Broken pipe
A 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 process 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, respectively, 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 respectively, 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).

Super-user
A process is recognized as a *super-user* 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.

File Name.

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 file names 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 file names.

Path Name and Path Prefix

A path name 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 file name.

More precisely, a path name is a null-terminated character string constructed as follows:

```
<path-name>::=<file-name>| <path-prefix><file-name>|/
<path-prefix>::=<rtprefix>| /<rtprefix>
<rtprefix>::=<dirname>/| <rtprefix><dirname>/
```

where <file-name> 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 path name 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 path name 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 path name searches. A process's root directory need not be the root directory of the root file system, and is determined by the *userid* entry in */etc/passwd*. The working directory for each process is determined either by cd(1) or chdir(2).

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 super-user.

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.

Otherwise, 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 a 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. **Msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **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 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 super-user.

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** and 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** and 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 a 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 awaiting this semaphore's semval to become greater than its current value. **Semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions.

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is 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 super-user.

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** and 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** and 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 a 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, **shm_dtime** is the time of the last *shmdt* operation, and **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 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 super-user.

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** and 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** and 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 path name 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 path name. [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, and 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 will be 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 super-user to use this call.

Path points to a path name 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 will fail if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

An attempt is being 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 path name 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.

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.

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. *Sbrk* clears only the page actually allocated, starting at a page boundary.

Brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

> Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)). [ENOMEM]
>
> Such a 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 path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with */*.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the path name. [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 path name 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 super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user 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 then 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 will fail and the file mode will be 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 super-user. [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 path name 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 effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Chown will fail and the owner and group of the named file will 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 super-user. [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 path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /.

The effective user ID of the process must be super-user 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 will fail and the root directory will remain unchanged if one or more of the following are true:

Any component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not super-user. [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*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*.

Close will fail 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 path name 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).
>
> The "save text image after execution bit" of the mode 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 80 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail 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 path name 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]
>
> Eighty (80) 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, namely 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*, *open*, *dup*, *fcntl*, or *pipe* 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* system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Eighty (80) file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion a non-negative integer, namely 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 path name 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;
```

and it 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 will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Except for SIGPHONE and SIGWIND, signals set to be caught by the calling process will be set to terminate 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 will not be 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 will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's path name 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 5120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

Path, *argv*, or *envp* point to an illegal address. [EFAULT]

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be 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 of 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 eight 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.

For each semaphore for which the calling process has set a 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)].

If the process ID, tty group ID, and process group ID of the calling process are equal, (i.e. it is a process group leader), the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

If the process is a process group leader, all processes in its group are made members of the *null* group.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

intro(2), semop(2), signal(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* 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 will remain open across *exec*, otherwise the file will be 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).

F_GETLK Get the first block which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(5)]. The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return immediately with an error value of −1.

F_SETLKW	This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure describes the type (*l_type*), starting offset (*l_start*), relative offset (*l_whence*), size (*l_len*), process id (*l_pid*), and system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of the file by setting *l_len* to zero (0). If such a lock also has *l_where* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file [see *chmod*(2)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

Fcntl will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[EINVAL]	*Cmd* is F_DUPFD. *Arg* is either negative, or greater than or equal to, the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*Cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[EACCES]	*Cmd* is F_SETLK, the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write lock (F_WRLCK) and the segment of a file to be locked is already read or write locked by another process.

[ENOLCK]	*Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked).
[EDEADLK]	*Cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling process to sleep, waiting for that lock to become free, would cause a deadlock.
[EFAULT]	*Cmd* is F_SETLK, *arg* points outside the program address space.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

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.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.

WARNINGS

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

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_ING**, 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)
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.

The child process has a different amount of time left until an alarm clock signal (see *alarm*(2)).

Fork will fail and no child process will be 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 writeups of various devices in Section 7 discuss how *ioctl* applies to them.

Ioctl will fail 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.

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 a process or a group of processes. The process or group of processes to which the signal is to be sent is 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 super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(2)) and will be referred to below as *proc0* and *proc1* respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be 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 super-user, *sig* will be 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 super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not –1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be 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 super-user, 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 path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link will fail and no link will be 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 super-user. [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 path name. [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*, *open*, *dup*, or *fcntl* 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 will fail and the file pointer will remain 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 path name 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 follow-
        ing
                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 super-user for file types other than FIFO special.

Mknod will fail and the new file will not be created if one or more of the following are true:

The process's effective user ID is not super-user. [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*. is the standard UNIX PC directory for mounting floppy diskettes. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will 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 **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

Mount will fail if one or more of the following are true:

The effective user ID is not super-user. [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

mount(1M), 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 super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*. Only super user 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 super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*.

Msgctl will fail 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 super user and it 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 super user. [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), stdipc(3C).

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 are 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 will fail 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, namely 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), stdipc(3C).

NAME

msgop – 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 will not be sent and the calling process will return immediately.
>
> If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend 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 will fail and no message will be 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 will return immediately with a return value of −1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend 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 will fail and no message will be 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* return 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), stdipc(3C).

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 will fail and not change the nice value if *incr* is negative and the effective user ID of the calling process is not super-user. [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 path name 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 will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be 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).

The "save text image after execution bit" of the mode 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* will fail 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 80 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]

Eighty (80) 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, namely a file descriptor, is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2).

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* will 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; with a return value of −1 from *pause* and *errno* 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 5120 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 will fail 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 super-user to use this call. *Op* specifies the following:

PROCLOCK –	lock text and 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 will fail and not perform the requested operation if one or more of the following are true:

The effective user ID of the calling process is not super-user. [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 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 will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), monitor(3C).

BUGS

Profil() is not supported on the UNIX PC.

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 the list), 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*. Also, the parent 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 *func*; see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will 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 (as on PDP-11s), 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 (as on the 3B-20 and VAX-11/780), either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests will 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 range from 0 to 1024. The *data* argument is ignored. This request will fail 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*. Request 4 writes a word into I space, and request 5 writes a word into D space. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *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:

the general registers (D0-D7, A0-A7)

certain bits of the Processor Status Word (all bits except SUPERVISOR state and interrupt level)

the PC register

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 and 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 will fail 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 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.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal **SIGTRAP**.

GENERAL ERRORS

Ptrace will in general fail 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 will return a 0.
>
> If O_NDELAY is clear, the read will block 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 will return a 0.
>
> If O_NDELAY is clear, the read will block until data becomes available.

Read will fail 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), window(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:*

GETVAL	Return the value of semval (see *intro*(2)). {READ}
SETVAL	Set the value of semval to *arg.val*. {ALTER} When this cmd is successfully executed the semadj value 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 semvals into array pointed to by *arg.array*. {READ}
SETALL	Set semvals 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 command can only be executed by a process that has an effective user ID equal to either that of super user 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 command can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

Semctl will fail 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 super user and it 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.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

semget(2), semop(2), stdipc(3C).

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 are 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 will fail 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 semaphore identifiers system wide would be exceeded. [ENOSPC]

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, namely 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), stdipc(3C).

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. The contents of 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:

> If *sem_op* is a negative integer, one of the following will occur: {ALTER}
>
> > If semval (see *intro*(2)) 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* will return immediately.
> >
> > If semval is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend 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 will occur: {READ}

If semval is zero, *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend 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 will fail 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 an **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

exec(2), exit(2), fork(2), semctl(2), semget(2), stdipc(3C).

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), window(7).

BUGS

Setpgrp cannot be called from processes associated with windows. Any process calling *setpgrp* must have stdin, stdout, and stderr directed to devices other than window devices to function properly.

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 super-user, 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 super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [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 super user 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 super user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

Shmctl will fail 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 super user and it 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), stdipc(3C).

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 *size* bytes (see *intro*(2)) are created for *key* if one of the following are 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 will fail 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, namely 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), stdipc(3C).

NAME

shmop – 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 will fail and 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 will fail and 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), stdipc(3C).

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)
SIGWIND	20	window status changes
SIGPHONE	21	telephone status changes

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 of 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) plus a "core image" will be 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* will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call will return 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 will fail 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]

SIGWIND and SIGPHONE are ignored by default and are reset to SIG.IGN upon an *exec*(2) system call.

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).

WARNING

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 UNIX, these signals will continue to behave as described below; they are included only for compatibility with other versions of UNIX. 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 of are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is SIGCLD, the calling process's child processes will 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 will be queued and the signal-catching function will be 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* will block until all of the calling process's child processes terminate; it will then return 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 will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding 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.

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 path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name 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*, *creat*, *dup*, *fcntl*, or *pipe* 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:

```
dev_t    st_dev;    /* ID of device containing */
                    /* a directory entry for this file */
ino_t    st_ino;    /* Inode number */
ushort   st_mode;   /* File mode; see mknod(2) */
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 */
dev_t    st_rdev;   /* ID of device */
                    /* This entry is defined only for */
                    /* character special or block */
                    /* special files */
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 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 will fail 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 will fail 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 will fail if the effective user ID of the calling process is not super-user. [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 i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

Syslocal – local system calls

SYNOPSIS

#include <sys/syslocal.h>

int syslocal (cmd [, arg] ...)

DESCRIPTION

Syslocal executes special AT&T UNIX PC system calls. *Cmd* is the name of one of the system calls described below.

SYSL_REBOOT Reboots the system. You must be superuser to execute. No additional arguments are required.

SYSL_KADDR Returns certain kernel addresses or values. This call is used by programs like *ps*(1) so that they don't have to read the kernel symbol table. The second argument is one of the following:

SLA_V	returns address of V
SLA_PROC	returns address of proc table
SLA_TIME	returns address of system time
SLA_USRSTK	returns top of user stack
SLA_USIGN	returns signature, unique # for each version
SLA_BLDDATE	returns address of build date string
SLA_BLDPWD	returns address of build directory string
SLA_MEM	returns size of physical memory
SLA_BDEVCNT	returns maximum number of block devices
SLA_CDEVCNT	returns maximum number of character devices

SYSL_LED Turns on/off user LED. The second argument is either 0 for off or 1 for on.

The following two calls support the hardware real-time clock. Their use requires the additional include file:

#include <sys/rtc.h>

SYSL_RDRTC Reads the real-time clock. The second argument is a *struct rtc ** .

SYSL_WRTRTC Writes the real-time clock. The second argument is a *struct rtc ** .

The following two calls support loadable device drivers. Their use requires the additional include file:

#include <sys/drv.h>

SYSL_ALLOCDRV	Allocates/deallocates space for a loadable driver and returns driver status. The second argument is one of the following:

DRVALLOC	allocates space
DRVUNALLOC	releases allocated space
DRVSTAT	returns driver status

The third argument is a *struct drvalloc ** You must be superuser to execute DRVALLOC and DRVUNALLOC.

SYSL_BINDDRV	Loads/unloads a loadable driver. The second argument is either DRVBIND for loading or DRVUNBIND for unloading. The third argument is a *struct drvbind ** . You must be superuser to execute.

The following two calls support installable fonts.

SYSL_LFONT	Installs a font.
SYSL_UFONT	Deinstalls a font.

In both cases, two arguments are required: the font file pathname (dummy pointer for SYSL_UFONT) and the font slot number (0 to 15). Again, you must be superuser to execute. See *window*(7) for additional font information.

Supplying a font slot number between 0 and 7 causes the font to be inherited at that slot number by all subsequent windows. Preloading fonts into slots 8-15 allows these fonts to be installed without going to the file system so they can be loaded rapidly. This is useful for applications which refer to more than 8 fonts because the font activity is more efficient.

If you attempt to load a font into a slot which is currently occupied, you will not get an error condition, but rather, the old font will be swapped out and the new one loaded in. You can also deinstall a font from slots 0 through 7, if the font to be deinstalled is not being accessed. If it is being accessed ERRNO is set to EBUSY.

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 will fail 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. The following is this contents of the structure:

```
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 on DEC processors, 100ths of a second on WECo processors.

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_utime*s and *tms_cutime*s of the child processes.

Tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

Times will fail if *buffer* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths (100ths) 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 super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user 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 path name. 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 super-user.

Umount will fail if one or more of the following are true:

The process's effective user ID is not super-user. [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 UNIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current UNIX 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 UNIX 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 UNIX is running on.

Uname will fail 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 path name pointed to be *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 super-user. [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 to 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 will fail 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 path name 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 super-user 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 will fail 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 super-user and not the owner of the file and *times* is not **NULL**. [EPERM]

The effective user ID is not super-user and not the owner of the file and *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 and 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 will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will 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 will be zero and the low order 8 bits will 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 will fail and return 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*, *open*, *dup*, *fcntl*, or *pipe* 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 flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain 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 was 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 will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes will be permitted. Thus, the write will fail 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) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will 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 UNIX 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. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3M) These functions constitute the Math Library, *libm*. They are automatically loaded as needed by the FORTRAN compiler. 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>**.

(3T) These functions constitute the UNIX PC "terminal access method" (tam) library.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may 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.

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>**.

FILES

/lib/libc.a
/lib/libm.a

SEE ALSO

ar(1), cc(1), ld(1), nm(1), intro(2), stdio(3S).

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. As 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 six 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 six characters, *a64l* will use the first six.

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

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

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *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* 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, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

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, 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

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 according as 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.

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 will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

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

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

This function is available only in the domestic (U.S.) version of the UNIX PC software.

Crypt is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

Key is a user's typed password. *Salt* is a two-character string chosen from the set [**a-z A-Z 0-9 .** /]; this string is used to perturb the DES algorithm in one of 4096 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 two 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. This is the key that will be 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(1M), passwd(1), getpass(3C), passwd(4).

BUGS

The return value points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>

char *ctermid(s)
char *s;

DESCRIPTION

Ctermid generates the path name 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 path name 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 will refer to the terminal if used as a file name. Thus *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 UNIX 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 three-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-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, like *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

curses – screen functions with "optimal" cursor motion

SYNOPSIS

cc [flags] files **–lcurses –ltermcap** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold,
termio(7) termcap(5)

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) coordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping

noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	change all of *win*
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win,c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
winsch(win,c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*
wstandend(win)	end standout mode on *win*
wstandout(win)	start standout mode on *win*

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)** will be 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 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 in *phone*(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 often important for certain 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).
phone(7), termio(7) in the *UNIX Administrator's Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the **<dial.h>** header file.

```
INTRPT    -1  /* interrupt occurred */
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.**

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

An *alarm*(2) system call for 3600 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 expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

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

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0.$$

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

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8.$$

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 thereto. 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 if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be 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 so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

SEE ALSO

brk(2), malloc(3C), stdio(3S).

NAME

eprintf – send a message to the status manager

SYNOPSIS

```
#include <status.h>
int eprintf (mtype, mact, uname, format [, arg ] ... )
int mtype, mact;
char *uname, *format;
```

DESCRIPTION

Eprintf formats the passed message a la *printf* and writes the message to the error device. The status manager wakes up whenever the error device is written to, queues the message, and displays an icon to indicate a message is waiting.

Mtype (message type) can have one of the following values:

ST_MAIL	Mail messages
ST_CAL	Calendar messages
ST_OTHER	Miscellaneous messages
ST_SYS	Kernel error messages
ST_LOG	Log message in log file
ST_POP	Popup message

Mact (message action) can have one of the following values:

ST_DISPLAY	Just display message
ST_EXEC	Execute process (message text is shell command line in this case)
ST_NOTIFY	Notify caller on display (sends caller SIGUSR1)
ST_CONFIRM	Signal caller with confirmation/denial on display (SIGUSR1 = Yes, SIGUSR2 = No)
ST_OFF	Remove messages from queue
ST_LOGFILE	Log message in log file

Uname points to the user login name that the message is for. The status manager will only display the message pending icon when this user is logged in. If *uname* is NULL (or if it points to a null string), then the message is displayed regardless of who is logged in.

ST_POP will cause the message to be acted on immediately, rather than displaying an icon and waiting for the user to click. ST_LOG will take the first word of the formatted message (i.e., up to the first space) as a file name, which it will open as a logfile in **/usr/adm**. The rest of the message will then be inserted in the file, followed by a time stamp.

DIAGNOSTICS

Eprintf returns −1 if error (open of error device failed).

SEE ALSO

message (3T), tam(3T).

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, 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 zero.

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise 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 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 (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;

DESCRIPTION

Fopen opens the file named by *file-name* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with the *stream*.

File-name 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 obtained from *open*, *dup*, *creat*, or *pipe*(2), which will open files but not return pointers to a FILE structure *stream* which are necessary input for many of the section 3S library routines. 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

form – display and accept forms

SYNOPSIS

```
#include <menu.h>
#include <form.h>
int form(form, op)
form_t *form;
int op;
```

DESCRIPTION

This routine manipulates a form as determined by the operation code (*op*). If the *op* arg is F_BEGIN, the form is initialized and displayed. If *op* is F_INPUT, user input is accepted. If *op* is F_END, the form is terminated and removed from the display. These functions may be combined in many ways. By specifying (F_BEGIN | F_INPUT | F_END), the caller creates a "pop-up" form which is initialized (displayed), used for input, then removed. Generally, (F_BEGIN | F_INPUT) is used for the first call, F_INPUT for each subsequent interaction, and F_END when the form is to be discarded.

During the F_INPUT function, the user may point to fields with the mouse or with the keyboard (arrows, Prev, Next, Beg, Home, End, Tab). The user may may modify fields by typing and editing (Back Space, Dlete Char, Clear Line, Cancel) or by selecting a choice from a menu optionally associated with the field.

The *form* structure has the following form:

```
typedef struct
{
        char        *f_label;     /* form label */
        char        *f_name;      /* form name */
        char        f_flags;      /* form flags */
        int         f_win;        /* form window */
        track_t     *f_track;     /* tracking info */
        field_t     *f_fields;    /* fields */
        field_t     *f_curfl;     /* current field */
} form_t;
```

F_label is the form label, displayed on the window label line of the form. If *f_label* is NULL, no label is displayed.

F_name is the form name, or NULL if the form has no name.

F_flags contains flags. The F_WINNEW flag causes *form* to use the "new" algorithm to place the window. Basically, the new algorithm looks for relatively empty screen space to place the window. F_WINSON causes *form* to use the "son" algorithm which causes the new window to slightly overlap the current window. If neither F_WINNEW nor F_WINSON is given, the "popup" algorithm is used. This causes the new window to appear near the middle of the current window, inside it if possible. F_NOMOVE is set if the

Move icon is not to be displayed on the border of the form. F_NOHELP is set if the Help icon is not to be displayed on the form border.

F_win holds the window identifier associated with this form. It is allocated on an F_BEGIN call, used on subsequent calls, and deleted on an F_END call. *F_track* is a pointer to the mouse-tracking information required during form interaction. The space for this data is allocated on F_BEGIN and freed on F_END.

F_fields points to the array of fields (see below). *F_curfl* points to the current field. The caller should point *f_curfl* to the default field. *Form* will modify *f_curfl* as the user moves the highlighting around in the form. The list of fields is terminated by a field whose *fl_name* is NULL.

Each field in the array pointed to by *f_fields* and *f_curcl* has the following form:

```
typedef struct
{
        char        *fl_name;     /* field name */
        char        fl_row;       /* field row  */
        char        fl_ncol;      /* name column */
        char        fl_fcol;      /* field column */
        char        fl_len;       /* field length */
        char        fl_flags;     /* flags */
        char        *fl_value;    /* field values */
        menu_t      *fl_menu;     /* assoc. menu pointer */
        char        *fl_prompt;   /* field prompt */
} field_t;
```

Fl_name is the field name. *Fl_row* is the row number on which to display the field. Row (and column) numbers are form-relative with 0,0 being the upper-leftmost location in the form. The form name (*f_name*) is located above 0,0 so the user needn't allocate a row for it.

Fl_ncol and *fl_fcol* control where the field name (*fl_ncol*) and field value (*fl_fcol*) are displayed. Generally, *fl_fcol* is greater than *fl_ncol* by at least the length of the field name.

Fl_len is the length of the field. See *fl_value*, below.

Fl_flags contains various flags which describe the field. F_CLEARIT specifies that any previous value for the field should be erased when the user tries to enter a new value. This is useful for fields where user editing makes little sense. F_MONLY means that the only allowable input to this field is via the associated menu (see *fl_menu*, below).

On call, *fl_value* contains the initial field value. On return, this string is modified to contain the user-supplied value. If no editing was performed by the user, the return value is the same as the call value. Note that the caller must supply a pointer to a character array at least *fl_len* + 1 bytes long. In addition, the caller should place a null byte after the end of the default value. For a 30 byte field, a default value might be of the form:

```
"Default Value\0              "
 1234567890123 45678901234567890
         1         2         3
```

Fl_menu points to an optional "associated menu." If the caller supplies a menu pointer, then the user may press the Cmd or Opts key on that field to invoke *menu*(3T) to parse the menu. The selected menu item's name (*mi_name*) is placed in the field's value (*fl_value*). If the F_MONLY flag is set for the field, then any attempt to edit the field's value will force the associated menu to pop-up. When a field has an associated menu, the SLECT and MARK keys step through the menu choices without displaying the menu.

The optional message pointed to by *fl_prompt* is displayed on the prompt line whenever the field is selected. As the user moves from field to field, the prompt changes.

EXAMPLE

The following program illustrates a typical use of *form*:

```
#include <tam.h>
#include <menu.h>
#include <form.h>
#include <stdio.h>
#include <kcodes.h>

mitem_t printitems[ ] =
{
        "ASR-33",               0,0,
        "Centronix",            0,1,
        "Diablo #1",            0,2,
        "Diablo #2",            0,3,
        "Epson in lab",         0,4,
        "Laser Printer",        0,5,
        "File",                 0,6,
        0,                      0,0
};

menu_t printmenu =
{
        "Printers",
        0,
        "Select a Printer from the list",
        0,1,0,0,
        M_SINGLE,
        {0},
        0,0,0,0,0,
        printitems,
        printitems,
        0
};

mitem_t priitems[ ] =
{
        "Low",          0,0
        "Normal",       0,1,
        "High",         0,2,
        "Immediate",    0,3,
        0,0,0,
};

menu_t primenu =
{
        "Printing Priority",
        0,
        "At what priority should the document be printed?",
        0,1,0,0,
        M_SINGLE,
        {0},
        0,0,0,0,0,
        priitems,
        &priitems[1],
        0
};

mitem_t yesnoitems[ ] =
{
        "No",   0,0,
        "Yes",  0,1,
        0,0,0
};

menu_t yesnomenu =
{
        0,
        0, "Select Yes (y) or No (n)",
        0,1,0,0,
        M_SINGLE,
        {0},
        0,0,0,0,0,
        yesnoitems,
        yesnoitems,
        0

};

field_t printfields [ ] =
{
        "Printer Name", 0,0,15,30,F_CLEARIT,
        "System Printer                 ",&printmenu,
        "Enter a Printer Name (touch CMD or OPTS to see
        choices)",
        "From Page",    1,0,15,5,0,
        "1      ",0,
        "Select the page number of the first page to be printed",
        "To Page",      1,25,40,5,0,
        "999    ",0,
        "Select the page number of the last page to be printed",

        "Priority",     2,0,15,10,F_MONLY,
        "Normal   ",&primenu,
        "Enter the print priority (Press CMD or OPTS to see
        choices)",

        "Delete After Printing?",       4,0,25,3,0,
        "No   ", &yesnomenu,
        "Do you wish the document to be deleted after it is
        printed?",

        0,                              0,0,0,0,0,
                0,0,
                0
};

form_t printform =
{
        "Print",
        "Printer Options",
        0,
        0,
        0,
        printfields,
        printfields
};



main()
{
        int err;
        int printop;
        char *which;

        winit();
        keypad(0,1);

        printop = M_BEGIN | M_INPUT;

        while(1)
        {
                which = "printform";
                err = form( &printform, printop );
                printop &= ~M_BEGIN;
                if ( err < 0 || err == Close )
                break;
        }
        if ( err < 0 )
        {
                fprintf(stderr,"fatal err in %s, code =
                %d",which,err);
                sleep(5);
        }
        wexit(0);
    }
```

FILES

/usr/include/form.h
/usr/include/menu.h
/usr/include/kcodes.h

SEE ALSO

menu(3T), tam(3T).

DIAGNOSTICS

Form returns non-negative keyboard codes (see **kcodes.h**) when keyboard input terminated the form interaction. Other return values signal more serious errors and are defined in **form.h**.

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, into an array beginning at *ptr*, *nitems* items of data from the named input *stream*, where 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, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

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 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

Although in UNIX an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non–UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such a 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 successfully be executed. 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* would be a file in a directory with read but without 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 some 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* will run 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.
It 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* will not have a chance to free that storage, so it will remain 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

gamma – log gamma function

SYNOPSIS

#include <math.h>

extern int signgam;

double gamma (x)
double x;

DESCRIPTION

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-negative 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

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 so 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 path-name of current working directory

SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

DESCRIPTION

Getcwd returns a pointer to the current directory path-name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *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.

SEE ALSO

environ(5).

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent – get group file entry

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 */
};
```

Getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, 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 and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it 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**>, which 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 to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Returns the **NULL** pointer if *name* 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>**, which causes it to increase the size of programs, not otherwise using standard I/O, 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 echoing. 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 will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpent, endpent – get and clean up printer status file entries

SYNOPSIS

```
#include <lp.h>

int getpent(p)
struct pstat *p;
int endpent()
```

DESCRIPTION

Getpent returns a structure describing a printer that is installed in the *lp* spooler subsystem. EOF is returned when no more printers are available.

Endpent is used to clean up after the last call to getpent.

```
struct pstat                          /* printer status entry */
{
      char    p_dest[DESTMAX+1];  /* destination name of printer */
      int     p_pid;               /* if busy, process id that is */
                                   /* printing, otherwise 0 */
      char    p_rdest[DESTMAX+1];/* if busy, the destination */
                                   /* requested by user at time of */
                                   /* request, otherwise "-" */
      int     p_seqno;             /* if busy, sequence # of */
                                   /* printing request */
      time_t  p_date;              /* date last enabled/disabled */
      char    p_reason[P_RSIZE];   /* if enabled, then "enabled" */
                                   /* otherwise the reason the */
                                   /* printer has been disabled. */
      short   p_flags;             /* See below for flag values. */
};

/* Value interpretation for p_flags: */

#define     P_ENAB 1               /* printer enabled */
#define     P_AUTO 2               /* disable printer automatically */
#define     P_BUSY 4               /* printer now printing a request */
```

FILES

These subroutines are located in the **libdev** library (**/usr/lib/libdev**).

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**>, which causes it to increase the size of programs, not otherwise using standard I/O, 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 returns 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;
};
```

This structure is declared in **<pwd.h>** so it is not necessary to redeclare it.

The *pw_comment* field is unused; the others have meanings described in *passwd*(4).

Getpwent when first called returns a pointer to the first **passwd** structure in the file; thereafter, it returns a pointer to the next **passwd** structure in the file; so 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 and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it 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>**, which 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

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 occurs, such as trying to use these functions on a file that has not been opened for reading, 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, then *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*. If the end of file is reached without a match, it fails.

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 expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *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 file. It is most often expected that 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 file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

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 contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would 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. The implicit read done by *pututline* if it finds that it isn't already at the correct place in the file will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, 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 non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

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, its source code has many other options available which the user may select 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 manner 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 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

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

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).

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* three-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 three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three 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* will fail 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* will close the file and free 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* will reinitialize **OFFSET(***ldptr***)** to the file address of the next archive member and return **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* will fail 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

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* will fail 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* will fail 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* will fail 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* will fail if *sectindx* is greater than the number of sections in the object file; *ldnlseek* will fail if there is no section name corresponding with **sectname*. Either function will fail 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* will fail 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**, then *ldopen* will open *filename* and allocate and initialize the **LDFILE** structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE**(*ldptr*) is the archive magic number, *ldopen* will reinitialize the **LDFILE** structure for the next archive member of *filename*.

Ldopen and *ldclose* are designed to work in concert. *Ldclose* will return **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* will open *filename* anew and allocate and initialize 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* will fail if *sectindx* is greater than the number of sections in the object file; *ldnrseek* will fail if there is no section name corresponding with *sectname*. Either function will fail 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 **libld.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* will fail if *sectindx* is greater than the number of sections in the object file; *ldnshread* will fail if there is no section name corresponding with *sectname*. Either function will fail 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* will fail if *sectindx* is greater than the number of sections in the object file; *ldnsseek* will fail if there is no section name corresponding with *sectname*. Either function will fail 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 will return the the index of the next entry.

Ldtbindex will fail 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* will fail 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), 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 return **SUCCESS** or **FAILURE**. *Ldtbseek* will fail 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

lockf – record locking on files

SYNOPSIS

#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

Fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order in order to establish lock with this function call.

Function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define  F_ULOCK  0   /* Unlock a previously locked section */
#define  F_LOCK   1   /* Lock a section for exclusive use */
#define  F_TLOCK  2   /* Test and lock a section for exclusive use */
#define  F_TEST   3   /* Test section for other process' locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

Size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a **−1** and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EBADF]
Fildes is not a valid open descriptor.

[EACCES]
Cmd is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK]
Cmd is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), read(2), write(2)

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

WARNINGS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O pacakage is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

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(1M), 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.