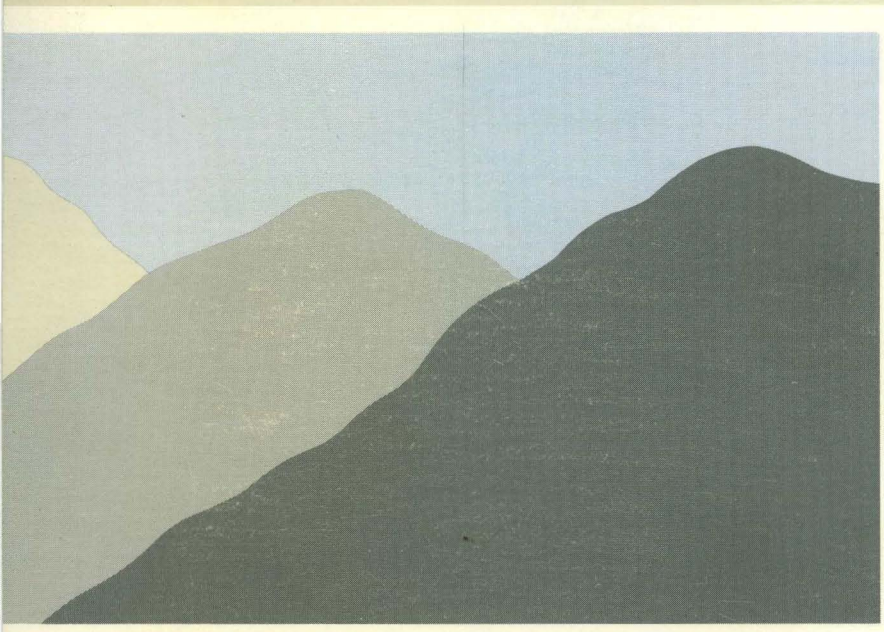

ROS Reference Manual (Section 1)

RIDGE

9010

Ridge Operating System (ROS)

Reference Manual

Ridge Computers
Santa Clara, CA

fourth edition: 9010-E (NOV 84)
second update: 9010-E2 (JUN 85)

PUBLICATION HISTORY

Manual Title: ROS Reference Manual

First Edition: 9010 (OCT 83)
Second Edition: 9010-A (DEC 83)
Third Edition: 9010-B (FEB 84)
Fourth Edition: 9010-E (NOV 84) ROS 3.2
first update: 9010-E1 (JAN 85)
second update: 9010-E2 (JUN 85) ROS 3.3

NOTICE

ACKNOWLEDGEMENT

This software and documentation is based in part on the fourth Berkeley Software Distribution, under license from the regents of the University of California. We acknowledge the following individuals for their part in its development: Ken Arnold, Rick Blau, Earl Cohen, Robert Corbett, John Foderaro, Charles Haley, Mark Horton, Bill Joy, Howard Katseff, Randy King, Jim Kleckner, J.E. Kulp, Steve Levine, Colin McMaster, Geoffrey Peck, Rob Pike, Eric Scott, Kurt Shoens, Eric Shienbrood, and David Wasley.

TRADEMARKS

PREFACE

This manual describes the features of the Ridge Operating System (ROS) in detail. ROS is derived from UNIX System V releases 1 and 2, the Berkeley Software Distribution (bsd) 4.1 and 4.2, and other sources.

The top center of each page bears the name of the source of each software entity. If the software entity was obtained from the Berkeley Software Distribution 4.1, for example, the heading reads **(bsd 4.1)**. If it was created by Ridge Computers, the heading reads **(Ridge)**.

The sections of this ROS Reference Manual are:

Commands and Application Utilities

(1) User and Administrator Commands

System Calls, Subroutines, Libraries

(2) System Calls for C

(3A) C-callable Subroutines
(3B) Pascal-callable Subroutines
(3C) Assembler and C Library Routines
(3M) Mathematical Library Routines
(3S) Standard I/O Package
(3X) Miscellaneous Routines

(4) File Formats

(5) Miscellaneous Facilities

(7) Device Characteristics

CONVENTIONS USED IN THIS MANUAL

All entries in the various sections are based on a common format, although all parts of the common format do not always appear.

The **NAME** part gives the name(s) of the entry and briefly states its purpose.

The **SYNTAX** part states the syntax of the command or routine. Here, certain conventions apply:

> **Boldface** strings are literals that must be typed just as they appear.
>
> *Italic* or *underlined* items usually represent parameters for which the user must substitute a value or name.
>
> Square brackets [] around an argument indicate that the enclosed argument is optional.
>
> Ellipses..indicate that the previous argument may be repeated any number of times.
>
> -, +, and = are flag characters for optional literal parameters.

The **DESCRIPTION** part discusses the subject at hand.

The **EXAMPLES** part gives examples of usage, where appropriate.

The **FILES** part gives the file names that are required for correct operation of the command or routine, and any temporary files created.

The **SEE ALSO** part gives references to related information. The number in the parentheses is the section number where the referenced command or routine is documented.

The **DIAGNOSTICS** part explains the error messages that may be generated, not including the self-explanatory ones.

The **WARNINGS** part points out potential pitfalls.

The **BUGS** part explains known bugs or deficiencies, and sometimes a suggested work-around.

TABLE OF CONTENTS

SECTION 1: COMMANDS

SECTION 2: C System Calls

SECTION 3B: Pascal-callable Subroutines

SECTION 3C: C-callable Subroutines

SECTION 3F: FORTRAN Library

SECTION 3S: Standard I/O Package

SECTION 3X, 3M: Math and Special Libraries

SECTION 4: File Formats

SECTION 5: Miscellaneous Facilities

SECTION 6: Games

SECTION 7: Special Files

PERMUTED INDEX

The permuted index is a sorted list of each keyword found in the manual entry titles. To find all manual entries related to a particular keyword, find that word in column two, which is ordered alphabetically. The information is found on the page named in the righthand column.

NAME

intro – introduction to commands and application programs

DESCRIPTION

Section One describes, in alphabetical order, publicly-accessible commands.

DIAGNOSTICS

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

NAME

admin – create and administer SCCS files

SYNTAX

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

DESCRIPTION

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

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

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

–n This keyletter indicates that a new SCCS file is to be created.

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

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

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

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

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

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

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

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

i Causes the "No id keywords (ge6)" message issued by *get*(1) or *delta*(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(1)) are found in the text retrieved or stored in the SCCS file.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

−z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **−h**, above).

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

FILES

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

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

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

sum. Another **admin −h** is recommended to ensure the SCCS file is valid.

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

SEE ALSO

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

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ANSITAR – a program to read and write ANSI labeled tapes

SYNTAX

ansitar [crxtvblnf] [blocksize] [labelsize] [filecount] file

DESCRIPTION

Ansitar is a program which reads and writes ANSI format labelled tapes. Its options are similar to tar:

c	create a tape
r	replace (update) a tape
x	extract files
t	print table of contents
v	set verbose mode: for crx, print names; for t print labels
0-31	select drive 0-31 (default 0)
b	use next argument as a block size (default is 2048 bytes)
B	use next argument as tape block size, and following argument as line size (for blocking/unblocking)
l	use next arg as a label size
U	select upper(out)/lower(in) case translation of names
R	use RT11 label and name conventions
S	use RSTS conventions (same as RT11 except 80-byte labels)
P	create/read files in "pip" (FILES-11) counted format.
D	read variable length records (D format)
f	use next argument as number of files to search for **t** or **x** options. Ignored if file names specified.

The filename may contain * to indicate a wild card; however, the asterisk should be enclosed in quotes to prevent the shell from interpreting it.

EXAMPLES

ansitar tv	Verbose table of contents
ansitar xvDUb 2000 file1 file2	Extract file1 and file2 from tape with 2000 bytes/block and variable length records
ansitar xv *.c	Extract all files with .c extension from tape with 2048 byte/block
ansitar tvf 5	List contents of first 5 files
ansitar xvf 5	Extract the first five files

FILES

/dev/mt??

SEE ALSO

tar(1), mt(7)

DIAGNOSTICS

Many and verbose. Most are self-explanatory.

bad block size: block size <= 0

bad line size: line size <= 0 or not divisor of block size

label size must be >= <int>: label size too small to hold header

can't open <tapefile>: couldn't open tape device

can't allocate buffer of <int> bytes: malloc couldn't get buffer

can't allocate line buffer of <int> bytes: malloc couldn't get buffer

can't create <file>

can't open <file>

BUGS

Writing variable length records (D format) is not implemented.

Wild cards have not been extensively tested.

AUTHORS

Purdue University
Spencer Thomas, University of Utah.
David Brown, Varian

NAME

apply – apply a command to a set of arguments

SYNTAX

apply [– **ac**] [– *n*] command args ...

DESCRIPTION

Apply runs the named *command* on each argument *arg* in turn. Normally arguments are chosen singly; the optional number *n* specifies the number of arguments to be passed to *command*. If *n* is zero, *command* is run without arguments once for each *arg*. Character sequences of the form *%d* in *command*, where *d* is a digit from 1 to 9, are replaced by the *d*'th following unused *arg*. If any such sequences occur, *n* is ignored, and the number of arguments passed to *command* is the maximum value of *d* in *command*. The character '*%*' may be changed by the – **a** option.

Examples:

```
apply echo *
```

is similar to ls(1);

```
apply – 2 cmp a1 b1 a2 b2 ...
```

compares the 'a' files to the 'b' files;

```
apply – 0 who 1 2 3 4 5
```

runs who(1) 5 times; and

```
apply 'ln %1 /usr/joe' *
```

links all files in the current directory to the directory /usr/joe.

SEE ALSO

sh(1)

BUGS

Shell metacharacters in *command* may have bizarre effects; it is best to enclose complicated commands in single quotes ' '.

There is no way to pass a literal '*%2*' if '*%*' is the argument expansion character.

NAME

apropos – locate commands by keyword lookup

SYNTAX

apropos keyword ...

DESCRIPTION

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

apropos password

and

apropos editor

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

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

FILES

/usr/lib/whatis data base

SEE ALSO

man(1), whatis(1)

NAME

ar – archive and library maintainer

SYNTAX

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

DESCRIPTION

Ar maintains an archive file that consists of groups of files. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose.

Afile is the archive file. The *names* are constituent files in the archive file. *Key* is one of the following: (Keys d, r, m, q, and s automatically invoke ranlib to add a table of contents named **_.symdef** to the beginning of the archive, making the use of ranlib on each archive unnecessary.)

d Delete the named files from the archive file.

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

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

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

p Print the named files in the archive.

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

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file. Normally the 'last-modified' date of each extracted file is the date when it is extracted. However, if **o** is used, the 'last-modified' date is reset to the date recorded in the archive.

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

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

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

s Symbol table. Regenerate the archive symbol table even if ar(1) is not invoked with a command which will modify the archive contents. This is useful after strip(1) has been used on the archive.

FILES

/tmp/v* temporaries

SEE ALSO

lorder(1), ld(1), ranlib(1), ar(4)

BUGS

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

The 'last-modified' date of a file will not be altered by the **o** option if the user is not the owner of the extracted file, or the super-user.

NAME

as – Ridge relocatable assembler

SYNTAX

as [– **n**] [– **l** listfile] [– **o** objfile] file-name.**S**

DESCRIPTION

As activates the Ridge instruction assembler. *As* takes a source input file and assembles it into a relocatable object file. If no source file is specified, the standard input device is used for input. If an input file is named, it must end with the **.s** or **.S** extension.

The default name of the output object file is the same as the input file, except with a **.o** extension replacing the **.s**. If input is read from the standard input device, output will be directed to standard output. The case of the **.o** extension matches that of the **.s** extension.

The following flags may be specified in any order:

– **n** specifies that upper- and lowercase characters not be distinguished (that they be considered the same) when comparing symbol names. By default, upper- and lowercase characters are considered distinct, or different.

– **l** *listfile*
The source file with line numbers, the corresponding hexadecimal object code, and a cross reference of symbol names is written to *listfile*. By default, no such listing is produced.

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

SEE ALSO

ld(1)

NAME

asa – interpret ASA carriage control characters

SYNTAX

asa [files]

DESCRIPTION

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

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

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

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

```
a.out |asa |lpr
```

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

```
asa file
```

SEE ALSO

fort(1), fsplit(1), ratfor(1).

NAME

at, batch – execute commands at a later time

SYNTAX

at *time* [*date*] [+ *increment*]
at -r [*job* ...]
at -l [*job* ...]

batch

DESCRIPTION

At and *batch* read commands from standard input and execute them *at* a later time. *At* allows you to specify when the commands should be executed, while jobs queued with *batch* will execute when system load level permits. **At -r** removes jobs previously scheduled with *at*. The **-l** option reports all jobs scheduled for the invoking user.

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

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

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

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

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

Thus legitimate commands include:

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

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

Batch submits a batch job. *Batch is almost equivalent to "at* but it goes into a different queue, and "at now" will respond with the error message "too late."

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

EXAMPLES

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

This sequence can be used at a terminal:

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

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

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

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

```
echo "sh shellfile" |at 1900 thursday next week
```

FILES

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

SEE ALSO

cron(1), kill(1), mail(1), nice(1), ps(1), sh(1).

DIAGNOSTICS

Complains about various syntax errors and times out of range.

NAME

awk – pattern scanning and processing language

SYNTAX

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

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as **–f** *file*. The *prog* string should be enclosed in single quotes (') to protect it from the shell.

Parameters, in the form x=... y=... etc., may be passed to *awk*.

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

An input line is made up of fields separated by white space. (This default can be changed by using FS, see below). The fields are denoted **$1**, **$2**, ...; **$0** refers to the entire line.

A pattern-action statement has the form:

```
pattern { action }
```

A missing action means print the line; a missing pattern always matches. An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next    # skip remaining patterns on this input line
exit    # skip the rest of the input
```

Statements are terminated by semicolons, new-lines, or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The **C** operators ++, ––, +=, –=, *=, /=, and %= are also available in expressions. Variables may be real numbers, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any numeric or character string; this allows for a form of associative memory. String constants are quoted (").

The *print* statement prints its arguments on the standard output (or on a file if *>expr* is present), separated by the current output field separator, and terminated by the output record separator. The *printf* statement formats its expression list according to the format (see *printf*(3S)).

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer; *substr*(*s*, *m*, *n*) returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf*(*fmt*, *expr*, *expr*, ...) formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

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

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

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

The special patterns BEGIN and END may be used to capture control before the first and after the last input line is read. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields by starting the program with:

```
BEGIN { FS = c }
```

or by using the **−F***c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default new-line); and OFMT, the output format for numbers (default **%.6g**).

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

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

Add up first column, print sum and average:

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

Print fields in reverse order:

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

Print all lines between start/stop pairs:

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

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

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

Print file, filling in page numbers starting at 5:

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

command line: awk −f program n=5 input

SEE ALSO

grep(1), lex(1), sed(1).
Awk—A Pattern Scanning and Processing Language in *ROS Utility Guide* .

BUGS

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

NAME

banner – make posters

SYNTAX

banner strings

DESCRIPTION

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

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

Each argument appears on a separate line of the output.

EXAMPLES

$ banner this is a test

```
this
is
a
test
```

$ banner 'this is a' 'test'

```
this is a
test
```

$ banner 'this is' 'a test of' 'banner'

```
this is
a test of
banner
```

NAME

basename, dirname – deliver portions of path names

SYNTAX

basename string [suffix]
dirname string

DESCRIPTION

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

Dirname delivers all but the last level of the path name in *string*.

EXAMPLES

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

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

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

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

SEE ALSO

sh(1).

BUGS

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

NAME

bc – arbitrary-precision arithmetic language

SYNTAX

bc [**–c**] [**–l**] [file ...]

DESCRIPTION

Bc is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The **–l** argument stands for the name of an arbitrary precision math library. The syntax for *bc* programs is as follows; L means letter a–z, E means expression, S means statement.

Comments

are enclosed in /* and */.

Names

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

Other operands

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

Operators

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

Statements

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

Function definitions

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

Functions in –l math library

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

All function arguments are passed by value.

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

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

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

EXAMPLE

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

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

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

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

FILES

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

SEE ALSO

dc(1).
BC–An Arbitrary Precision Desk-Calculator Language in *ROS Utility Guide* .

BUGS

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

NAME

bdiff – big diff

SYNTAX

bdiff file1 file2 [n] [– **s**]

DESCRIPTION

Bdiff is used in a manner analogous to *diff*(1) to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for *diff*. *Bdiff* ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. The value of *n* is 3500 by default. If the optional third argument is given, and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail. If *file1* (*file2*) is – , the standard input is read. The optional – **s** (silent) argument specifies that no diagnostics are to be printed by *bdiff* (note, however, that this does not suppress possible exclamations by *diff*. If both optional arguments are specified, they must appear in the order indicated above.

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

FILES

/tmp/bd?????

SEE ALSO

diff(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

bfs – big file scanner

SYNTAX

bfs [–] name

DESCRIPTION

The *Bfs* command is (almost) like *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 512 characters, including new-line, per line (255 for 16-bit machines). *Bfs* is usually more efficient than *ed* for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

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

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

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

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

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

xo [*file*]
: Further output from the **p** and null commands is diverted to the named *file*, which, if necessary, is created mode 666. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

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

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

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

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

that the command

```
xb/^/ label
```

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

xt *number*

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

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

The variable name is the specified *digit* following the **xv**. The commands **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. The command **Xv61,100p** assigns the value **1,100p** to the variable **6**. To reference a variable, put a **%** in front of the variable name. For example, using the above assignments for variables **5** and **6**:

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

will all print the first 100 lines.

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

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

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

could be used to match and list lines containing *printf* of characters, decimal integers, or strings.
Another feature of the **xv** command is that the first line of output from a UNIX system command can be stored into a variable. The only requirement is that the first character of *value* be an **!**. For example:

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

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

```
xv7\!date
```

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

xbz *label*

xbn *label*

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

string **size**.

```
xv55
: l
/size/
xv5!expr %5 - 1
!if 0%5 != 0 exit 2
xbn l
xv45
: l
/size/
xv4!expr %4 - 1
!if 0%4 = 0 exit 2
xbz l
```

xc [*switch*]

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

SEE ALSO

csplit(1), ed(1), regcmp(3X).

DIAGNOSTICS

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

NAME

binmail – send mail to users or read mail

SYNTAX

binmail [**–epqr**] [**–f** file]

binmail [**–t**] persons

DESCRIPTION

Binmail is the older mail program originally supplied with Version 7. *Binmail*(1) is now linked to *Mail* and *mail* to provide a more capable mail interface. *Binmail* is only necessary for local mail delivery. *Binmail* without arguments prints a user's mail, message-by-message, in last-in, first-out order. For each message, the user is prompted with a **?**, and a line is read from the standard input to determine the disposition of the message:

<new-line>	Go on to next message.
+	Same as <new-line>.
d	Delete message and go on to next message.
p	Print message again.
–	Go back to previous message.
s [*files*]	Save message in the named *files* (**mbox** is default).
w [*files*]	Save message, without its header, in the named *files* (**mbox** is default).
m [*persons*]	Mail the message to the named *persons* (yourself is default).
q	Put undeleted mail back in the *mailfile* and stop.
EOT (control-d)	Same as **q**.
x	Put all mail back in the *mailfile* unchanged and stop.
!*command*	Escape to the shell to do *command*.
*****	Print a command summary.

The optional arguments alter the printing of the mail:

–e causes mail not to be printed. An exit value of 0 is returned if the user has mail; otherwise, an exit value of 1 is returned.

–p causes all mail to be printed without prompting for disposition.

–q causes *binmail* to terminate after interrupts. Normally an interrupt only causes the termination of the message being printed.

–r causes messages to be printed in first-in, first-out order.

–f*file* causes *binmail* to use *file* (e.g., **mbox**) instead of the default *mailfile*.

When *persons* are named, *binmail* takes the standard input up to an end-of-file (or up to a line consisting of just a **.**) and adds it to each *person*'s *mailfile*. The message is preceded by the sender's name and a postmark. Lines that look like postmarks in the message, (i.e., "From ...") are preceded with a >. The **–t** option causes the message to be preceded by all *persons* the mail is sent to. A *person* is usually a user name recognized by *login*(1). If a *person* being sent mail is not recognized, or if *binmail* is interrupted during input, the file **dead.letter** will be saved to allow editing and resending.

To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1C)). Everything after the first exclamation mark in *persons* is interpreted by the remote system. In particular, if *persons* contains additional exclamation marks, it can denote a sequence of machines through which the message is to be sent on the way to its ultimate destination. For example, specifying **a!b!cde** as a recipient's name causes the message to be sent to user **b!cde** on system **a**. System **a** will interpret that destination as a request to send the message to user **cde** on system **b**. This might be useful, for instance, if the sending system can access system **a** but not system **b**, and system **a** has access to system **b**.

The *mailfile* may be manipulated in two ways to alter the function of *binmail*. The *other* permissions of the file may be read-write, read-only, or neither read nor write to allow different levels of privacy. If changed to other than the default, the file will be preserved even when empty to perpetuate the desired permissions. The file may also contain the first line:

Forward to *person*

which will cause all mail sent to the owner of the *mailfile* to be forwarded to *person*. This is especially useful to forward all of a person's mail to one machine in a multiple machine environment.

When a user logs in, the presence of mail, if any, is indicated. Also, notification is made if new mail arrives while using *binmail*.

FILES

/bin/binmail	
/etc/passwd	to identify sender and locate persons
/usr/mail/*user*	incoming mail for *user*; i.e., the *mailfile*
$HOME/mbox	saved mail
$MAIL	variable containing path name of *mailfile*
/tmp/ma*	temporary file
/usr/mail/*.lock	lock for mail directory
dead.letter	unmailable text

SEE ALSO

login(1), mail(1), sendmail(1), uucp(1), write(1).

BUGS

Race conditions sometimes result in a failure to remove a lock file.
After an interrupt, the next message may not be printed; printing may be forced by typing a **p**.
Does not call *sendmail*(1). Provides local and uucp mail only.

NAME

bs – a compiler/interpreter for modest-sized programs

SYNTAX

bs [file [args]]

DESCRIPTION

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

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

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

```
statement
label statement
```

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

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

Statement Syntax:

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

break
Break exits from the inner-most *for/while* loop.

clear
Clears the symbol table and compiled statements. *Clear* is executed immediately.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

! shell command
An immediate escape to the Shell.

...
This statement is ignored. It interjects commentary in a program.

Expression Syntax:

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

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

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

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

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

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

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

(False, True)[a == b]

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

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

– expression
The result is the negation of the expression.

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

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

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

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

Binary Operators (in increasing precedence):

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

–
_ (underscore) is the concatenation operator.

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

< <= > >= == !=

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

+ -

Add and subtract.

*** / %**

Multiply, divide, and remainder.

^

Exponentiation.

Built-in Functions:

Dealing with arguments

arg(i)

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

narg( )

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

Mathematical

abs(x)

is the absolute value of x.

atan(x)

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

ceil(x)

returns the smallest integer not less than x.

cos(x)

is the cosine of x (radians).

exp(x)

is the exponential function of x.

floor(x)

returns the largest integer not greater than x.

log(x)

is the natural logarithm of x.

rand( )

is a uniformly distributed random number between zero and one.

sin(x)

is the sine of x (radians).

sqrt(x)

is the square root of x.

String operations

size(s)

the size (length in bytes) of s is returned.

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

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

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

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

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

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

File handling

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

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

Examples are given in the following section.

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

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

Tables

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

item(name, i)

key()

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

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

iskey(name, word)

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

Odds and ends

eval(s)

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

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

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

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

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

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

plot(request, args)

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

Call	*Function*
plot(0, term)	causes further *plot* output to be piped into *tplot*(1G) with an argument of – T*term*.
plot(4)	"erases" the plotter.
plot(2, string)	labels the current point with *string*.
plot(3, x1, y1, x2, y2)	draws the line between (*x1*,*y1*) and (*x2*,*y2*).
plot(4, x, y, r)	draws a circle with center (*x*,*y*) and radius *r*.
plot(5, x1, y1, x2, y2, x3, y3)	draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).

plot(6)	is not implemented.
plot(7, x, y)	makes the current point (*x*,*y*).
plot(8, x, y)	draws a line from the current point to (*x*,*y*).
plot(9, x, y)	draws a point at (*x*,*y*).
plot(10, string)	sets the line mode to *string*.
plot(11, x1, y1, x2, y2)	makes (*x1*,*y1*) the lower left corner of the plotting area and (*x2*,*y2*) the upper right corner of the plotting area.
plot(12, x1, y1, x2, y2)	causes subsequent x (y) coordinates to be multiplied by *x1* (*y1*) and then added to *x2* (*y2*) before they are plotted. The initial scaling is **plot(12, 1.0, 1.0, 0.0, 0.0)**.

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

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

PROGRAMMING TIPS

Using *bs* as a calculator:

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

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

The outline of a typical *bs* program:

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

Input/Output examples:

```
#    Copy "oldfile" to "newfile".
open("read", "oldfile", "r")
open("write", "newfile", "w")
...
while ?(write = read)
...
# close "read" and "write":
close("read")
close("write")

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

SEE ALSO

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

NAME

cal – print calendar

SYNTAX

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that used by England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January.
Note that "cal 78" refers to the first century A.D., not the 20th.

NAME

calendar – reminder service

SYNTAX

calendar [–]

DESCRIPTION

Calendar consults the file **calendar** in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates such as "Dec. 7," "december 7," "12/7," etc., are recognized, but not "7 December' or "7/12". On weekends, "tomorrow" extends through Monday.

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

FILES

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

SEE ALSO

mail(1).

BUGS

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

NAME

cat – concatenate and print files

SYNTAX

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

DESCRIPTION

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

cat file

prints the file, and:

cat file1 file2 >file3

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

If no input file is given, or if the argument – is encountered, *cat* reads from the standard input file. Output is buffered unless the – **u** option is specified. The – **s** option makes *cat* silent about non-existent files.

The – **v** option causes non-printing characters (with the exception of tabs, new-lines and form-feeds) to be printed visibly. Control characters are printed ^*X* (control-*x*); the DEL character (octal 0177) is printed ^**?**. Non-ASCII characters (with the high bit set) are printed as M-*x*, where *x* is the character specified by the seven low order bits.

When used with the – **v** option, – **t** causes tabs to be printed as ^**I**'s, and – **e** causes a $ character to be printed at the end of each line (prior to the new-line). The – **t** and – **e** options are ignored if the – **v** option is not specified.

WARNING

Do not **cat** a file to itself, such as

cat file1 file2 >file1

because the contents of **file1** will be lost. (The destination file would be opened and cleared before the two source files are **cat**'ed, thus ruining **file1** before it is read.)

SEE ALSO

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

NAME

cb – C program beautifier

SYNTAX

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

DESCRIPTION

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

SEE ALSO

cc(1).

The C Programming Language by B. W. Kernighan and D. M. Ritchie.

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cc, pcc – C invoker

SYNTAX

cc [option] ... file1 [file2] ...

DESCRIPTION

Cc compiles, optimizes, assembles, and links programs yielding executable code.

file.**c** is assumed to a C source file. Each **.c** file is compiled, and the resulting object file is given the name of the source with **.o** substituted for the **.c**. The **.o** file is normally removed, however, if a single C program is compiled and loaded in one go.

file.**s** is assumed to be an assembly source file. Each **.s** file is assembled and put into an object file named *file*.o.

The following options are interpreted by *cc* and *pcc*. See *ld*(1) for link editor options and *cpp*(1) for more preprocessor options.

– **a** Do not automatically convert operands of type float to type double in arithmetic expressions. This results in using single-precision arithmetic operators when evaluating expressions with operands of type float, and double-precision operators when evaluating expressions with operands of type double. Any module using functions declared with float or double formal parameters that is compiled without this option actually expects doubles. Before passing floats to stdio and math library functions, cast them to type double.

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

– **p** (NOT CURRENTLY IMPLEMENTED) Arrange for the compiler to produce code which counts the number of times each routine is called; also, if link editing takes place, replace the standard startoff routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

– **f** allow the C program to call Pascal and FORTRAN routines

– **O** Invoke an object-code optimizer.

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

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

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

– **B***string*

Construct pathnames for substitute compiler, assembler and link editor passes by concatenating *string* with the suffixes **cpp**, **c0** (or **ccom** or **comp**, see under FILES below), **c1**, **c2**, **as** and **ld**. If *string* is empty it is taken to be **/lib/o**.

– **t**[**p012al**]

Find only the designated compiler, assembler and link editor passes in the files whose names are constructed by a – **B** option. In the absence of a – **B** option, the *string* is taken to be **/lib/n**. – **t** "" is equivalent to – **tp012**.

– W*c,arg1[,arg2...]*

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

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

FILES

file.c	input file
file.o	object file
a.out	linked output
/tmp/ctm*	temporary
/lib/cpp	C preprocessor *cpp*(1)
/lib/ccom	Ridge C compiler, *cc*
/lib/c2	optional optimizer
/lib/oc*	(a name strating with "oc") backup compiler, *occ*
/lib/nc*	(a name starting with "nc") test compiler, *ncc*
/bin/as	assembler, *as*(1)
/bin/ld	link editor, *ld*(1)
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling (NOT CURRENTLY AVAILABLE)
/lib/libc.a	standard library, see (3)

SEE ALSO

The C Programming Language by B. W. Kernighan and D. M. Ritchie.
Ridge C Programming Notes (9014).

adb(1), cpp(1), as(1), ld(1), prof(1), sdb(1), monitor(3C).

NAME

cd – change working directory

SYNTAX

cd [directory]

DESCRIPTION

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

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

SEE ALSO

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

NAME

cdc – change the delta commentary of an SCCS delta

SYNTAX

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

DESCRIPTION

Cdc changes the *delta commentary*, for the *SID* specified by the –**r** keyletter, of each named SCCS file.

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

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

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

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

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

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

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

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

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

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

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

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

unescaped new-line character terminates the *comment* text.

The exact permissions necessary to modify the SCCS file are documented in the *Source Code Control System User's Guide*. Simply stated, they are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

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

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

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

does the same thing.

WARNINGS

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

FILES

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

SEE ALSO

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

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

cflow – generate C flow graph

SYNTAX

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

DESCRIPTION

Cflow analyzes a collection of C, YACC, LEX, and assembler files and attempts to build a graph charting the external references. Files with the **.y**, **.l**, **.c**, and **.i** extensions are YACC'd, LEX'd, and C-preprocessed (bypassed for **.i** files) as appropriate and then run through the first pass of *lint*(1). (The – **I**, – **D**, and – **U** options of the C-preprocessor are also understood.) Files with the **.s** extension are assembled and information is extracted from the symbol table. A graph of all external references is produced and displayed on the standard output.

Each output line consists of a reference number (a line number), appropriate indentation to indicate its level, the name of the global (normally only a function not defined as an external or beginning with an underscore; see below for the – **i** inclusion option), a colon, and its definition. For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char** *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the file name and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

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

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

```
int     i;

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

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

the command

```
cflow file.c
```

produces the the output

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

When the nesting level becomes too deep, the – **e** option of *pr*(1) can be used to compress the

tab expansion to something less than every eight spaces.

The following options are interpreted by *cflow*:

- **r** Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.
- **ix** Include external and static data symbols. The default is to include only functions in the flow graph.
- **i_** Include names that begin with an underscore. The default is to exclude these functions (and data if *-ix* is used).
- **d**num The *num* decimal integer indicates the depth at which the flow graph is cut off. By default this is a very large number. Do not set the cutoff depth to a nonpositive integer.

DIAGNOSTICS

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

SEE ALSO

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

BUGS

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

NAME

chfn – change finger entry

SYNTAX

chfn [loginname]

DESCRIPTION

Chfn changes the information about users. **Chfn** modifies the "optional" entry for the calling user in the **/etc/passwd**. The "optional" field usually contains the user's real-life name, office room number, office phone number, and home phone number. **Chfn** prompts the user fort each field. Included in the prompt is a default value, which is enclosed between brackets. The default value is accepted simply by typing <return>. To enter a blank field, type the word 'none'. Below is a sample run:

Name [Biff Studsworth II]:
Room number (Exs: 597E or 197C) []: 521E
Office Phone (Ex: 1632) []: 1863
Home Phone (Ex: 987532) [5771546]: none

Chfn allows phone numbers to be entered with or without hyphens. **Chfn** insists upon a four digit number (after the hyphens are removed) for office phone numbers. Also, room numbers must be in Evans or Cory.

The super-user may specify *loginname* to change the data for a specific user.

FILES

/etc/passwd, /etc/ptmp

SEE ALSO

finger(1), passwd(4)

BUGS

The office information is relevant only at the University of California at Berkeley. The office and phone number information format is constrained for use at Berkeley.

For historical reasons, the user's name, etc are stored in the passwd file. This is a bad place to store the information.

Because two users may try to write the passwd file at once, a synchronization method was developed. On rare occasions, a message that the password file is "busy" will be printed. In this case, *chfn* sleeps for a while and then tries to write to the passwd file again.

NAME

chmod – change mode

SYNTAX

chmod mode files

DESCRIPTION

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

4000	set user ID on execution
2000	set group ID on execution
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic *mode* has the form:

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

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

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

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

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter **s** is only useful with **u** or **g** and **t** only works with **u**. Only the owner of a file (or the super-user) may change its mode.

EXAMPLES

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

```
chmod o- w file
chmod +x file
```

SEE ALSO

ls(1), chmod(2).

NAME

chown, chgrp – change owner or group

SYNTAX

chown owner file ...

chgrp group file ...

DESCRIPTION

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

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

FILES

/etc/passwd
/etc/group

SEE ALSO

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

NAME

chsh – change default login shell

SYNTAX

chsh name [shell]

DESCRIPTION

Chsh changes the shell that is invoked upon logging in. **Chsh** modifies the login-shell entry for user *name* in the **/etc/passwd** file. If no *shell* is specified, **/bin/sh** is used.

Only the super-user can specify a shell besides **sh** and **csh**.

EXAMPLE

```
$ chsh bill /bin/csh
```

SEE ALSO

csh(1), passwd(1), passwd(4)

NAME

clear – clear terminal screen

SYNTAX

clear

DESCRIPTION

Clear clears your screen.

It looks in the environment for the terminal type (TERM) and uses curses(3X) and terminfo(4) to figure out the control character for clearing the screen.

NAME

cmp – compare two files

SYNTAX

cmp [– **l**] [– **s**] file1 file2

DESCRIPTION

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

Options:

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

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

SEE ALSO

comm(1), diff(1).

DIAGNOSTICS

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

NAME

comb – combine SCCS deltas

SYNTAX

comb [–**o**] [–**s**] [–**p**sid] [–**c**list] files

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the 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 generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

–**p***SID* The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

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

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

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

100 * (original – combined) / original

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

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

FILES

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

SEE ALSO

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

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

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

NAME

comm – select or reject lines common to two sorted files

SYNTAX

comm [– [**123**]] file1 file2

DESCRIPTION

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

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm – 12** prints only the lines common to the two files; **comm – 23** prints only lines in the first file but not in the second; **comm – 123** is a no-op.

SEE ALSO

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

NAME

compact, uncompact, ccat – compress and uncompress files, and cat them

SYNTAX

compact [name ...]
uncompact [name ...]
ccat [file ...]

DESCRIPTION

Compact compresses the named files using an adaptive Huffman code. If no file names are given, the standard input is compacted to the standard output. *Compact* operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to prepend a decoding tree to the compressed file since the encoder and the decoder start in the same state and stay synchronized. Furthermore, *compact* and *uncompact* can operate as filters. In particular,

... |compact |uncompact |...

operates as a (very slow) no-op.

When an argument *file* is given, it is compacted and the resulting file is placed in *file.C; file* is unlinked. The first two bytes of the compacted file code the fact that the file is compacted. This code is used to prohibit recompaction.

The amount of compression to be expected depends on the type of file being compressed. Typical values of compression are: Text (38%), Pascal Source (43%), C Source (36%) and Binary (19%). These values are the percentages of file bytes reduced.

Uncompact restores the original file from a file compressed by *compact.* If no file names are given, the standard input is uncompacted to the standard output.

Ccat cats the original file from a file compressed by *compact,* without uncompressing the file.

RESTRICTION

The last segment of the filename must contain fewer than thirteen characters to allow space for the appended '.C'.

FILES

*.C compacted file created by compact, removed by uncompact

SEE ALSO

Gallager, Robert G., 'Variations on a Theme of Huffman', *I.E.E.E. Transactions on Information Theory,* vol. IT-24, no. 6, November 1978, pp. 668 - 674.

pack(1)

NAME

copyfloppy – Copy one floppy disc to another

SYNTAX

copyfloppy [-C] [-F]

DESCRIPTION

Copyfloppy reads an enitre Ridge floppy and duplicates the contents on another floppy. COPYFLOPPY reads the contents of one floppy and pauses to allow the user to remove the source floppy and insert the destination floppy.

-c causes files to be moved toward the begining of the destination floppy, consolidating unused space near the end.

-f causes the destination floppy to be put in Ridge floppy format during the duplicating, to avoid use of the ZERO command on the destination floppy.

To "crunch" the files on a floppy (relocate them into consecutive blocks near the beginning of the floppy), use *copyfloppy -c*. When the system pauses for the second floppy to be inserted, leave the source floppy in the drive.

NAME

cp, ln, mv – copy, link or move files

SYNTAX

cp file1 [file2 ...] target
ln file1 [file2 ...] target
mv file1 [file2 ...] target

DESCRIPTION

File1 is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same (take care when using *sh*(1) metacharacters). If *target* is a directory, then one or more files are copied (linked, moved) to that directory.

If *mv* determines that the mode of *target* forbids writing, it will print the mode (see *chmod*(2)) and read the standard input for one line (if the standard input is a terminal); if the line begins with **y**, the move takes place; if not, *mv* exits.

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

SEE ALSO

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

BUGS

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

Ln will not link across file systems.

NAME

cpio – copy file archives in and out

SYNTAX

cpio –o [**acBv**]

cpio –i [**BcdmrtuvfsSb6R**] [patterns]

cpio –p [**adlmruv**] directory

DESCRIPTION

Cpio –o (copy out) reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information.

Cpio –i (copy in) extracts files from the standard input which is assumed to be the product of a previous **cpio –o**. Only files with names that match *patterns* are selected. *Patterns* are given in the name-generating notation of *sh*(1). In *patterns*, meta-characters **?**, *****, and [...] match the slash **/** character. Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is ***** (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below.

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

The meanings of the available options are:

a Reset access times of input files after they have been copied.

B Input/output is to be blocked 5,120 bytes to the record (override the default 512 bytes per block) (does not apply to the *pass* option; meaningful only with data directed to or from magnetic tape.)

d *Directories* are to be created as needed.

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

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

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

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

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

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

H Make file headers compatible with DEC VAX-11 and PDP-11. Use with **cpio –i** to read a VAX-11 or PDP-11 cpio tape that was not created with **–c**.

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

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

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

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

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

6 Process an old (i.e., UNIX System *Sixth* Edition format) file. Only useful with **–i** (copy in).

R Interactive replace of newer files. A "y" response to the prompt will replace a newer file with an older one, any other response will not. Has no effect when used with the **-u**

option.

EXAMPLES

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

```
ls | cpio -o >/dev/mt0
cd olddir
find . -depth -print | cpio -pdl newdir
```

The trivial case

```
find . -depth -print | cpio -oB >/dev/rmt0
```

can be handled more efficiently by:

```
find . -cpio /dev/rmt0
```

SEE ALSO

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

NOTES

When transferring files between a Ridge 32 and a machine with different byte ordering (VAX, PDP-11, Zilog Z8000), use the **-c** compatibility option.

BUGS

Path names are restricted to 128 characters. If there are too many unique linked files, the program runs out of memory to keep track of them and, thereafter, linking information is lost.

NAME

cpp – the C language preprocessor

SYNTAX

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

DESCRIPTION

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

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

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

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

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

– **U***name*

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

operating system:	unix
hardware:	ridge

– **D***name*

– **D***name*=*def*

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

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

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

All *cpp* directives start with lines begun by #. The directives are:

#define *name token-string*

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

#define *name*(*arg*, ..., *arg*) *token-string*

Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma separated tokens, and a) by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma separated list.

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

#include *"filename"*
#include *<filename>*
Include at this point the contents of *filename* (which will then be run through *cpp*). When the *<filename>* notation is used, *filename* is only searched for in the standard places. See the **–I** option above for more detail.

#line *integer-constant "filename"*
Causes *cpp* to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. If *"filename"* is not given, the current file name is unchanged.

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

#ifdef *name*
The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

#ifndef *name*
The lines following will not appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

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

#else Reverses the notion of the test directive which matches this directive. So if lines previous to this directive are ignored, the following lines will appear in the output. And vice versa.

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

FILES

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

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

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

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

NAME

cron - clock daemon

SYNTAX

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times. Regularly scheduled commands can be specified according to instructions found in crontab files; users can submit their own crontab file via the *crontab* command. Commands which are to be executed only once may be submitted via the *at* command. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file **/etc/rc.**

Cron only examines crontab files and at command files during process initialization and when a file changes. This reduces the overhead of checking for new or changed files at regularly scheduled intervals.

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/log	accounting information
/usr/spool/cron	spool area

SEE ALSO

at(1), crontab(1), rc(5), sh(1)

DIAGNOSTICS

A history of all actions taken by cron are recorded in **/usr/lib/cron/log.**

NAME

crontab - user crontab file

SYNTAX

crontab [*file*]
crontab -r
crontab -l

DESCRIPTION

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

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

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

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

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

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

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

NOTE: Users should remember to redirect the standard output and standard error of their commands! If this is not done, any generated output or errors will be mailed to the user.

FILES

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

SEE ALSO

cron(1), sh(1).

NAME

crypt – encode/decode

SYNTAX

crypt [password]

DESCRIPTION

Crypt encrypts and decrypts files according to a password.

Crypt reads from the standard input and writes on the standard output.

The *password* is a key that selects a particular transformation. If no *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

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

will print the clear.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* in encryption mode.

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

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

To discourage a spy from trying all combination of keywords to read your file, the transformation of files is deliberately slow. But a keyword of three lowercase characters would be quite easy to crack in a short time, so make your keywords longer.

To prevent your keyword from showing up in the command field of the ps(1) output, **crypt** destroys the record of the keyword immediately upon execution.

The choice of keys and key security are the most vulnerable aspect of *crypt*.

You may encryt a file twice, with two different keywords.

According to federal export regulations, **crypt** is not available in the International versions of the Ridge Operating System.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(1).

BUGS

If output is piped to *nroff* and the encryption key is *not* given on the command line, *crypt* can leave terminal modes in a strange state (see *stty*(1)).
If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

NAME

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

SYNTAX

csh [**–cefinstvVxX**] [arg ...]

DESCRIPTION

Csh is a first implementation of a command language interpreter incorporating a history mechanism (see **History Substitutions**) job control facilities (see **Jobs**) and a C-like syntax.

An instance of *csh* begins by executing commands from the file '.cshrc' in the *home* directory of the invoker. If this is a login shell then it also executes commands from the file '.login' there. It is typical for users on crt's to set terminal options by putting a *stty*(1) command "stty crt" in their *.login* file.

In the normal case, the shell will then begin reading commands from the terminal, prompting with '% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

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

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

Lexical structure

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

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

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

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ';' and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an '&'.

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

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with '&', the shell prints a line which looks like:

[1] 1234

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else, you may hit the key ^Z (control-Z) which sends a STOP signal to the current job. The shell will then normally indicate that the job has been 'Stopped', and print another prompt. You can then manipulate the state of this job, putting it in the background with the *bg* command, or run some other commands and then eventually bring the job back into the foreground with the foreground command *fg*. A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. Another special key is ^Y, which does not generate a STOP signal until a program attempts to *read*(2) it. The ^Y key is useful when you have prepared some commands for a job which you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "stty tostop". If you set this tty option, then background jobs will stop when they try to produce output.

There are several ways to refer to jobs in the shell. The character '%' introduces a job name. If you wish to refer to job number 1, you can name it as '%1'. Just naming a job brings it to the foreground; thus '%1' is a synonym for 'fg %1', which brings job 1 back into the foreground. Similarly '%1 &' returns job 1 to the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous. Thus '%ex' would normally restart a suspended *ex*(1) job, if there were only one suspended job whose name began with the string 'ex'. It is also possible to say '%?string' which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a '+' and the previous job with a '−'. The abbreviation '%+' refers to the current job and '%−' refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), '%%' is also a synonym for the current job.

Status reporting

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable *notify*, the shell will notify you immediately of changes of status in background jobs. There is also a shell command *notify* which marks a single process so that any change in its status will be immediately reported. By default, *notify* marks the current process; simply say 'notify' after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that 'You have stopped jobs.' You may use the *jobs* command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

Substitutions

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

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence. History substitutions begin with the character '!' and may begin **anywhere** in the input stream (with the proviso that they **do not** nest.) This '!' may be preceded by an '\' to prevent its special meaning. For convenience, a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. (History substitutions also occur when an input line begins with '↑'. This special abbreviation will be described later.) Any input line which contains history substitution is echoed on the terminal before it is executed.

Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The size of which is controlled by the *history* variable; the previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

For example, consider the following output from the *history* command:

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

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

The current event is number 13. We can refer to previous events by event number. For example, '!11' refers to the same event as '!−2'. You can also refer to previous events by the prefix of a command word, as in '!d' for event 12 or '!wri' for event 9, or by a string contained in a word in the command as in '!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '!!' refers to the previous command; thus '!!' alone is essentially a *redo*.

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

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

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

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing '.xxx' component, leaving the root name.
e	Remove all but the extension '.xxx' part.
s/*l*/*r*/	Substitute *l* for *r*
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g. 'g&'.
p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further substitutions.
x	Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a 'g' the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

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

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

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

Quotations with ' and "

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

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

Alias substitution

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

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

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

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

Variable substitution

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

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

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

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

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

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

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

$name
${name}

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

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

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

$#name
${#name}
Gives the number of words in the variable. This can be used later in a '[selector]'.

$0
Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}
Equivalent to '$argv[number]'.

$*
Equivalent to '$argv[*]'.

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

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

$?name
${?name}
Substitutes the string '1' if name is set, '0' if it is not.

$?0
Substitutes '1' if the current input filename is known, '0' if it is not.

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

$<
Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script.

Command and filename substitution

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

Command substitution

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

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

Filename substitution

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

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

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

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

Input/output

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

< name

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

<< word

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

> name

>! name

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

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

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

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

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have, by default, no access to the text of the commands; rather they receive the original standard input of the shell. The '<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. Note that the default standard input for a command is **not** modified to be the empty file '/dev/null'; rather the standard input remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see **Jobs** above.)

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

Expressions

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

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

Here the precedence increases to the right, '==' '!=' '=~' and '!~', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '-', '*' '/' and '%' being, in groups, at the same level. The '==' '!=' '=~' and '!~' operators compare their arguments as strings; all others operate on numbers. The operators '=~' and '!~' are like '!=' and '==', except that the right hand side is a *pattern* (containing, e.g. '*'s, '?'s and instances of '[...]') against which the left hand operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings which begin with '0' are considered octal numbers. Null or missing arguments are considered '0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they should be surrounded by spaces.

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

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

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

Control flow

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

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

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

Builtin commands

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

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

bg
bg %job ...
Puts the current or specified jobs into the background, continuing them if they were stopped.

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

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

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

cd
cd name
chdir
chdir name

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

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

continue

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

default:

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

dirs

Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.

echo wordlist
echo **–n** wordlist

The specified words are written to the shells standard output, separated by spaces, and terminated with a newline unless the **–n** option is specified.

else
end
endif
endsw

See the description of the *foreach, if, switch,* and *while* statements below.

eval arg ...

(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) executed in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions.

exec command

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

exit
exit(expr)

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

fg
fg %job ...

Brings the current or specified jobs into the foreground, continuing them if they were stopped.

foreach name (wordlist)
...
end

The variable *name* is successively set to each member of *wordlist* and the sequence of commands between this command and the matching *end* are executed. (Both *foreach* and *end*

must appear alone on separate lines.)

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

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

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

history
history *n*
history **−r** *n*
history **−h** *n*
Displays the history event list; if *n* is given, only the *n* most recent events are printed. The **−r** option reverses the order of printout to be most recent first rather than oldest first. The **−h** option causes the history list to be printed without leading numbers. This is used to produce files suitable for sourcing using the −h option to *source*.

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

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

jobs
jobs **−l**
Lists the active jobs; given the −l options lists process id's in addition to the normal information.

kill %job
kill −sig %job ...
kill pid
kill −sig pid ...
kill **−l**
Sends either the TERM (terminate) signal or the specified signal to the specified jobs or

processes. Signals are either given by number or by names (as given in */usr/include/signal.h,* stripped of the prefix "SIG"). The signal names are listed by "kill −l". There is no default, saying just 'kill' does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal as well.

login
Terminate a login shell, replacing it with an instance of **/bin/login.** This is one way to log off, included for compatibility with *sh*(1).

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

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

nohup
nohup command
The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with '&' are effectively *nohup'ed.*

notify
notify %job ...
Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

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

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

popd
popd +n
Pops the directory stack, returning to the new top directory. The argument '+*n*' discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd
pushd name
pushd +n
With no arguments, *pushd* exchanges the top two elements of the directory stack. When a *name* argument is given, *pushd* changes to the new directory (ala *cd*) and pushes the old

current working directory (as in *csw*) onto the directory stack. When a numeric argument is given, *pushd* rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

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

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

set
set name
set name=word
set name[index]=word
set name=(wordlist)
The first form of the command shows the value of all shell variables. Variables which have other than a single word as their value appear as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word.* The fourth form sets the *index'th* component of name to word; this component must already exist. The final form sets *name* to the list of words in *wordlist.* In all cases, the value is expanded as described in the *Command and filename substitution* section.

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

setenv name value
Sets the value of environment variable *name* to be *value,* a single string. The most commonly used environment variable USER, TERM, and PATH are automatically imported to and exported from the *csh* variables *user, term,* and *path;* there is no need to use *setenv* for these.

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

source name
source **–h** name
The shell reads commands from *name. Source* commands may be nested; if they are nested too deeply, the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Normally input during *source* commands is not placed on the history list; the –h option causes the commands to be placed in the history list without being executed.

stop
stop %job ...
Stops the current or specified job which is executing in the background.

suspend
Causes the shell to stop in its tracks, much as if it had been sent a stop signal with ^**Z**.

This is most often used to stop shells started by *su*(1).

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

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

time
time command

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

umask
umask value

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

unalias pattern

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

unhash

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

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by 'unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be *unset*.

unsetenv pattern

Removes all variables whose name match the specified pattern from the environment. See also the *setenv* command above and *printenv*(1).

wait

All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

while (expr)
...

end

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

%job

Brings the specified job into the foreground.

%job &

Continues the specified job in the background.

@

@ name = expr

@ name[index] = expr

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

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

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

Pre-defined and environment variables

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

This shell copies the environment variable USER into the variable *user,* TERM into *term,* and HOME into *home,* and copies these back into the environment whenever the normal shell variables are reset. The environment variable PATH is handled in the same way. Once the path is set in the *.cshrc* file, inferior *csh* processes will import the definition of *path* from the environment, and re-export it if you then change it.

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

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

cwd The full pathname of the current directory.

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

histchars Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character !. The second character of its value replaces the character ↑ in quick substitutions.

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

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

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

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

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

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

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

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

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

notify If set, the shell notifies asynchronously of job completions. The default is to present job completions just before printing a prompt.

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

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

savehist This variable is given a numeric value to control the number of entries of the history list that are saved in ~/.history when the user logs out. Any command which has been referenced in this many events will be saved. During start up, the shell sources ~/.history into the history list, enabling history to be saved across logins. Large *savehist* values will slow down the shell during start up.

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

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

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will, upon termination, print a line giving user, system, and real times, along with a utilization percentage (which is the ratio of user plus system times to real time).

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

Non-builtin command execution

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

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

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

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

Argument list processing

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

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

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

−f The shell will start faster, because it will neither search for nor execute commands from the file

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

−n Commands are parsed, but not executed. This aids in syntactic checking of shell scripts.

−s Command input is taken from the standard input.

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

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

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

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

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

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

Signal handling

The shell normally ignores *quit* signals. Jobs running detached (either by '&' or the *bg* or *%... &* commands) are immune to signals generated from the keyboard, including hangups. Other signals have the same values the shell inherited from its parent. The way the shell handles interrupts and terminate signals in shell scripts can be controlled by *onintr.* Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file '.logout'.

AUTHOR

University of California, Berkeley

FILES

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

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 4096 characters. The number of arguments to a command that involve filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of *alias* substitutions on a single line to 20.

SEE ALSO

sh(1), access(2), execve(2), fork(2), kill(2), pipe(2), signal(2), umask(2), wait(2), termio(7), a.out(4), environ(5), 'An introduction to the C shell' in the **ROS Utility Guide.**

BUGS

When a command is restarted from a stop, the shell prints the directory it which it started if different from the current directory. The directory printed will be incorrect if the job has changed directories internally.

Shell builtin functions are not stoppable/restartable. Command sequences of the form 'a ; b ; c' are also not handled gracefully when stopping is attempted. If you suspend 'b', the shell will then immediately execute 'c'. This is especially noticeable if this expansion results from an *alias*. You can place the sequence of commands in ()'s to force it to a subshell, i.e. '(a ; b ; c)'.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. If a better virtual terminal interface existed, things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by '?', are not placed in the *history* list. Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

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

NAME

csplit – context split

SYNTAX

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

DESCRIPTION

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

00: From the start of *file* up to (but not including) the line referenced by *arg1*.
01: From the line referenced by *arg1* up to the line referenced by *arg2*.
:
:
n+ 1: From the line referenced by *argn* to the end of *file*.

The options to *csplit* are:

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

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

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

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

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or – some number of lines (e.g., **/Page/– 5**).

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

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

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

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

EXAMPLES

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

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

cat cobol0[0– 3] > file

Note that this example overwrites the original file.

csplit – k file 100 {99}

This example would split the file at every 100 lines, up to 10,000 lines. The – **k** option causes

the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

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

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

SEE ALSO

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

DIAGNOSTICS

Self explanatory except for:

arg – out of range

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

NAME

ctags - create a tags file

SYNTAX

ctags [-u] [-w] [-x] name...

DESCRIPTION

Ctags makes a tags file for the *ex*(1) editor from the specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these function definitions.

-x causes *ctags* to produce a list of function names, the line number and file name on which each is defined, and print the text of that line on the standard output. This is a simple index which can be printed out as an off-line readable function index.

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

Other options are:

– **w** suppressing warning diagnostics

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

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

FILES

tags	output tags file

SEE ALSO

ex(1), vi(1)

BUGS

Recognition of **functions, subroutines,** and **procedures** for FORTRAN and Pascal is done in a simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name, ctags doesn't see it.

NAME

ctrace – C program debugger

SYNTAX

ctrace [options] [file]

DESCRIPTION

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

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

The only *options* you will commonly use are:

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

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

– **o** Octal
– **x** Hexadecimal
– **u** Unsigned
– **e** Floating point

These *options* are used only in special circumstances:

– **l** *n* Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.
– **s** Suppress redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.
– **t** *n* Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The Diagnostics section explains when to use this option.
– **P** Run the C preprocessor on the input before tracing it. You can also use the **-D**, **-I**, and **-U** *cc*(1) preprocessor options.

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

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

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

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

EXAMPLE

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

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

and you enter these commands and test data:

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

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

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

the output will be:

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

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

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

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

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

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

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

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

EXECUTION-TIME TRACE CONTROL

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

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

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

You can also call these functions from *dbx*(1) if you compile with the **-g** option.

You can also turn the trace off and on by setting static variable tr_ct_ to 0 and 1, respectively.

DIAGNOSTICS

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

Ctrace Diagnostics

warning: some variables are not traced in this statement

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

warning: statement too long to trace

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

cannot handle preprocessor code, use -P option

This is usually caused by #ifdef/#endif preprocessor statements in the middle of a C

statement, or by a semicolon at the end of a #define preprocessor statement.

'if ... else if' sequence too long
Split the sequence by removing an **else** from the middle.

possible syntax error, try -P option
Use the **-P** option to preprocess the *ctrace* input, along with any appropriate **-D**, **-I**, and **-U** preprocessor options. If you still get the error message, check the Warnings section below.

Cc Diagnostics

warning: floating point not implemented
warning: illegal combination of pointer and integer
warning: statement not reached
warning: sizeof returns 0
Ignore these messages.

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

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

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

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

expression causes compiler loop: try simplifying
This is caused by a bug in the UNIX/370 C compiler. Unfortunately, the only way to avoid it is to use the *ctrace* **-v** option to not trace the function containing this line.

WARNINGS

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

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

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

BUGS

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

Pointer values are always treated as pointers to character strings.

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

FILES

runtime.c	run-time trace package

SEE ALSO

signal(2), ctype(3C), fflush(3S), longjmp(3C), printf(3S), setjmp(3C), string(3C).

NAME

cu – call another system

SYNTAX

cu [**–s** speed] [**–l** line] [**–h**] [**–t**] [**–d**] [**–m**] [**–o** |**–e**]

DESCRIPTION

Cu calls a UNIX System, a terminal, or a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files. *Speed* gives the transmission speed (110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200).

–l specifies a device name for the communications line device to be used.

–h emulates local echo, allowing calls to other computer systems which expect terminals to be in half-duplex mode. The **–t** option is used when dialing an ASCII terminal which has been set to auto-answer. Appropriate mapping of carriage-returns to carriage-return-line-feed pairs is set. The **–m** option specifies a direct line which has modem control. The **–e** (**–o**) option designates that even (odd) parity is to be generated for data sent to the remote.

After making the connection, *cu* runs as two processes. The *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system. The *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output. Normally, an automatic DC3/DC1 protocol controls input from the remote so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following:

~.	terminate the conversation.
~!	escape to an interactive shell on the local system.
~!*cmd...*	run *cmd* on the local system (via **sh –c**).
~$*cmd...*	run *cmd* locally and send its output to the remote system.
~% take *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~% put *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places.
~~*...*	send the line ~*...* to the remote system.
~% nostop	turn off the DC3/DC1 input control protocol for the remainder of the session. Use this if the remote system is one which does not respond to the DC3 and DC1 protocol.
~% b	or
~% break	transmit a BREAK to the remote system.
~^Z	suspend (only when using csh).

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

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

Data from the remote is diverted (or appended, if >> is used) to file. The trailing ~> terminates the diversion.

The use of **~% put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of **~% take** requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, **stty tabs** mode should be set on the remote system if tabs are to be copied without expansion.

FILES

/dev/null
/dev/tty?

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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

NAME

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

SYNTAX

cut – **c**list [file1 file2 ...]
cut – **f**list [– **d**char] [– **s**] [file1 file2 ...]

DESCRIPTION

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

The meanings of the options are:

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

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

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

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

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

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

HINTS

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

EXAMPLES

cut – d: – f1,5 /etc/passwd mapping of user IDs to names

name=`who am i |cut – f1 – d" "` to set **name** to current login name.

DIAGNOSTICS

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

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

no fields The *list* is empty.

SEE ALSO

grep(1), paste(1).

NAME

cxref – generate C program cross reference

SYNTAX

cxref [options] files

DESCRIPTION

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

In addition to the **– D**, **– I** and **– U** options (which are identical to their interpretation by *cc*(1)), the following *options* are interpreted by *cxref*:

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

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

– o file Direct output to named *file*.

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

– t Format listing for 80-column width.

FILES

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

SEE ALSO

cc(1).

DIAGNOSTICS

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

NAME

date – print and set the date

SYNTAX

date [[yy]mmddhhmm[.ss]] [+format]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set. The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *yy* is the last 2 digits of the year number and is optional; *ss* is the current second and is also optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

If the argument begins with +, the output of *date* is under the control of the user. The format for the output is similar to that of the first argument to *printf*(3S). All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by % and will be replaced in the output by its corresponding value. A single % is encoded by %% All other characters are copied to the output without change. The string is always terminated with a new-line character.

Field Descriptors:

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

EXAMPLE

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

would have generated as output:

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

DIAGNOSTICS

No permission	if you aren't the super-user and you try to change the date;
bad conversion	if the date set is syntactically incorrect;
bad format character	if the field descriptor is not recognizable.

WARNING

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

NAME

dbx – debugger

SYNTAX

dbx [**–r**] [**–i**] [**–I** *dir*] [*objfile*]

DESCRIPTION

Dbx source-level debugger and executor of cc(1) and f77(1) object files. If the cc(1) and f77(1) option **–g** is used, debugging may involve symbolic names. If **–g** is omitted, only the machine-level facilities of *dbx* can be used.

If no *objfile* is specified, **./a.out** is used. The object file contains a symbol table, which includes the name of all source files translated by the compiler to create it. These files are available for perusal while using the debugger.

If the file **.dbxinit** exists in the current directory, the debugger commands in it are executed in order. *Dbx* also checks for a **.dbxinit** in the user's home directory if there isn't one in the current directory.

The options are:

–r Execute *objfile* immediately. If it terminates normally, *dbx* exits. Otherwise, report the reason for termination and offer the user the option of entering the debugger or letting the program fault. *Dbx* reads from **/dev/tty** when **–r** is specified and standard input is not a terminal.

–i Force *dbx* to act as though standard input is a terminal.

–I *dir* Add *dir* to the list of directories that are searched when looking for a source file. Normally, *dbx* looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set with the **use** command.

Unless **–r** is specified, *dbx* just prompts and waits for a command.

Execution and Tracing Commands

run [*args*] [< *filename*] [> *filename*]
Start executing *objfile*, passing *args* as command line arguments; < or > can be used to redirect input or output in the usual manner. If *objfile* has been written since the last time the symbolic information was read in, *dbx* reads in the new information.

trace [**in** *procedure/function*] [**if** *condition*]
trace *source-line-number* [**if** *condition*]
trace *procedure/function* [**in** *procedure/function*] [**if** *condition*]
trace *expression* **at** *source-line-number* [**if** *condition*]
trace *variable* [**in** *procedure/function*] [**if** *condition*]
Trace during execution. A number is associated with the command that is used to turn the tracing off (see the **delete** command).

The first argument describes what is to be traced. If it is a *source-line-number*, then the line is printed immediately before being executed. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon, e.g. "mumble.c":17.

If the argument is a procedure or function name then every time it is called,

information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted, and if it's a function then the value it is returning is also printed.

If the argument is an *expression* with an **at** clause then the value of the expression is printed whenever the identified source line is reached.

If the argument is a variable then the name and value of the variable is printed whenever it changes. Execution is substantially slower during this form of tracing.

TRACE with no arguments causes every line to print before execution. This is pretty slow going.

The clause "**in** *procedure/function*" restricts tracing information to be printed only while executing inside the given procedure or function.

Condition is a boolean expression that is evaluated before printing the trace information.

stop if *condition*
stop at *source-line-number* [**if** *condition*]
stop in *procedure/function* [**if** *condition*]
stop *variable* [**if** *condition*]
Stop execution when the given line is reached, procedure or function called, variable changed, or condition true.

status [> *filename*]
Print out the currently active **trace** and **stop** commands.

delete *command-number* [,*command-number* ...]
Remove the trace or stop corresponding to *command-number*. The numbers associated with traces and stops are printed by the **status** command.

catch *number*
ignore *number*
Start or stop trapping signal *number* before it is sent to the program. This is useful when a program being debugged handles signals such as interrupts. Initially all signals are trapped except SIGCONT, SIGCHILD, SIGALRM and SIGKILL.

cont Continue execution from where it stopped. Execution cannot be continued if the process has "finished", that is, called the standard procedure "exit". *Dbx* does not allow the process to exit, thereby letting the user to examine the program state.

step Execute one program line.

next Execute up to the next source line, even if there is a procedure or function call between here and there.

Displaying and Naming Data

print *expression* [, *expression* ...]
Print out the values of the expressions. Array expressions are always subscripted by brackets ("[ ]"). Variables having the same identifier as one in the current block may be referenced as "*block-name . variable*". The field reference operator (".") can be used with pointers as well as records, making the C operator "->" unnecessary (although it is supported). The construct *typename*(*expression*) can be used to print the *expression*

out in the format of the named *type*.

whatis *name*
Print the declaration of the given name, which may be qualified with block names as above.

which *identifier*
Print the full qualification of the given identifier, i.e. the outer blocks that the identifier is associated with.

whereis *identifier*
Print the full qualification of all the symbols whose name matches the given identifier. The order in which the symbols are printed is not meaningful.

assign *variable* = *expression*
set *variable* = *expression*
Assign the value of the expression to the variable.

set *address* = *expression*
Assign the value of the *expression* to *address*.

set r*n* = *expression*
Assign the value of the *expression* to machine register *n*.

call *procedure(parameters)*
Execute the object code associated with the named procedure or function. Currently, calls to a procedure with a variable number of arguments are not possible. Also, string parameters are not passed properly for C.

where Print out a list of the active procedures and function.

dump [> *filename*]
Print the names and values of all active variables.

Accessing Source Files

edit [*filename*]
edit *procedure/function-name*
Invoke an editor on *filename*. Edit the current source file if *filename* is omitted. If a *procedure* or *function* name is specified, edit the file that contains it. The shell environment variable EDITOR specifies the editor to be invoked.

file [*filename*]
Change the current source file name to *filename*. If *filename* is omitted, print the current source file name.

func [*procedure/function*]
Change the current function. If none is specified then print the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

list [*source-line-number* [, *source-line-number*]]
list *procedure/function*
List the lines in the current source file from the first line number to the second inclusive. If no lines are specified, the next 10 lines are listed. If the name of a procedure or function is given lines *n*-*k* to *n*+*k* are listed where *n* is the first statement in the procedure or function and *k* is small.

use *directory-list*
Set the list of directories to be searched when looking for source files.

Machine-Level Commands

dr Display machine registers, program counter, and process id number.

tracei [*address*] [**if** *cond*]
tracei [*variable*] [**at** *address*] [**if** *cond*]
stopi [*address*] [**if** *cond*]
stopi [**at**] [*address*] [**if** *cond*]
Turn on tracing or set a stop using a machine instruction address.

stepi

nexti Single step as in **step** or **next**, but do a single instruction rather than source line.

address ,*address*/ [*mode*]
[*address*] / [*count*] [*mode*]
Print the contents of memory starting at the first *address* and continuing up to the second *address* or until *count* items are printed. If no address is specified, the address following the one printed most recently is used. The *mode* specifies how memory is to be printed; if it is omitted the previous mode specified is used. The initial mode is "X". The following modes are supported:

i	print the machine instruction
d	print a short word in decimal
D	print a long word in decimal
o	print a short word in octal
O	print a long word in octal
x	print a short word in hexadecimal
X	print a long word in hexadecimal
b	print a byte in octal
c	print a byte as a character
s	print a string of characters terminated by a null byte
f	print a single precision real number
g	print a double precision real number

Symbolic addresses are specified by preceding the name with an "&". Registers are denoted by "$rN" where N is the number of the register. Addresses may be expressions made up of other addresses and the operators "+", "-", and indirection (unary "*").

Miscellaneous Commands

sh [*command-line*]
Pass the command line to the shell for execution. The SHELL environment variable determines which shell is used.

alias *new-command-name old-command-name*
Respond to *new-command-name* as though it were *old-command-name*. Use this to make inconveniently long command names shorter.

help Print a synopsis of *dbx* commands.

source Read *dbx* commands from the given *filename*. Especially useful when the *filename* has been created by redirecting a **status** command from an earlier debugging session.

cnv *integer*
convert *number* to decimal, hex, and octal. *Number* can be any of those.

address *address*
print the line number corresponding to *address*.

address at *line-num*
print the address corresponding to *line-num*.

quit Exit *dbx*.

FILES

a.out object file

SEE ALSO

cc(1), f77(1)

BUGS

Non-local **gotos** (setjmp and longjmp) can cause some trace/stops to be missed.

At a C **switch** statement, the **step** command may take you to the bottom of the **switch**. Another **step** will take you to the proper case in the **switch**.

At an f77 computed **goto**, the **step** command may take you to the bottom of the computed **goto**. Another **step** will take you to the computed destination line.

Dbx suffers from a "multiple include" malady. If you have a program consisting of a number of object files and each is built from source files that include header files, the symbolic information for the header files is replicated in each object file. Since about one debugger start-up is done for each link, having the linker (ld) re-organize the symbol information won't save much time, though it would reduce some of the disk space used. The problem is an artifact of the unrestricted semantics of C **#include** statements. For example, an include file can contain static declarations that are separate entities for each file in which they are included.

If you **step** to enter a function or subroutine, an accurate stack traceback may not be possible until another **step** is used. Use **stop in** *function* to avoid this problem.

If Dbx leaves a trap instruction in the executable file, the program must be re-linked.

stop in *function* sometimes requires the function name to be qualified with the name of the file in which it appears. To stop in function **fun** of file **file.c**, the syntax might have to be **stop in file.fun**.

NAME

dc – desk calculator

SYNTAX

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
The value of the number is pushed on the stack. A number is an unbroken string of the digits 0– 9. It may be preceded by an underscore (_) to input a negative number. Numbers may contain decimal points.

+ – / * % ^
The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged. **P** interprets the top of the stack as an ASCII string, removes it, and prints it.

f All values on the stack are printed.

q exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x treats the top element of the stack as a character string and executes it as a string of *dc* commands.

X replaces the number on the top of the stack with its scale factor.

[...] puts the bracketed ASCII string onto the top of the stack.

<*x* >*x* =*x*
The top two elements of the stack are popped and compared. Register *x* is evaluated if they obey the stated relation.

v replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! interprets the rest of the line as a UNIX System command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. **I** pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O pushes the output base on the top of the stack.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

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

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

EXAMPLE

This example prints the first ten values of n!:

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

SEE ALSO

bc(1), which is a preprocessor for *dc* providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

x is unimplemented
where *x* is an octal number.

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

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

Out of headers
for too many numbers being kept around.

Out of pushdown
for too many items on the stack.

Nesting Depth
for too many levels of nested execution.

NAME

dd – convert and copy a file

SYNTAX

dd [option=value] ...

DESCRIPTION

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

option	*values*
if=*file*	input file name; standard input is default
of=*file*	output file name; standard output is default
ibs=*n*	input block size *n* bytes (default 4096)
obs=*n*	output block size (default 4096)
bs=*n*	set both input and output block size, superseding *ibs* and *obs*; also, if no conversion is specified, it is particularly efficient since no in-core copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
seek=*n*	seek *n* records from beginning of output file before copying
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to *ibs*
..., ...	several comma-separated conversions

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

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

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

SEE ALSO

cp(1).

DIAGNOSTICS

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

BUGS

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

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

NAME

debug – program debugging utility

SYNTAX

debug command argument ...

DESCRIPTION

When preceded by **debug**, any command or user program will be created and suspended. The user can then set breakpoints in it, and examine and/or modify it. For example:

```
$ debug myprog
```

or

```
$ debug pwd
```

The result is:

```
debug (18-Jun-83): process 000015D0 suspended
debug:
```

Now, type **h** for help:

```
debug: h
```

Debug cannot be run without a program name.

SEE ALSO

Ridge Debug Users Manual

NAME

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

SYNTAX

delta [–**r**SID] [–**s**] [–**n**] [–**g**list] [–**m**[mrlist]] [–**y**[comment]] [–**p**] files

DESCRIPTION

Delta permanently introduces into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

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

Delta may issue prompts on the standard output, depending upon certain keyletters specified and flags (see *admin*(1)) that may be present in the SCCS file (see –**m** and –**y** keyletters below).

Keyletter arguments apply independently to each named file.

–**r***SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *gets* for editing (**get** –**e**) on the same SCCS file were done by the same person (login name). The SID value specified with the –**r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(1)). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

–**s** Suppresses the issue of the created delta's SID on the standard output; as well as the number of lines inserted, deleted and unchanged in the SCCS file.

–**n** Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

–**g***list* Specifies a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

–**m**[*mrlist*] If the SCCS file has the **v** flag set (see *admin*(1)) then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

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

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

Note that if the **v** flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, *delta* terminates (it is assumed that the MR numbers were not all valid).

–**y**[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

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

−p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied in a *diff*(1) format.

FILES

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

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

WARNINGS

Lines beginning with an SOH ASCII character (binary 001) cannot be placed in the SCCS file unless the SOH is escaped. This character has special meaning to SCCS (see *sccsfile*(5)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (−) is specified on the *delta* command line, the **−m** (if necessary) and **−y** keyletters *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

SEE ALSO

admin(1), bdiff(1), cdc(1), get(1), help(1), prs(1), rmdel(1), sccsfile(4).
Source Code Control System User's Guide in *ROS Utility Guide.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

df – disc free

SYNTAX

df [-a] [*filesystems*]

DESCRIPTION

Df prints statistics for file systems. By default, statistics on all filesystems are reported.

dev	kbytes	used	avail	avg.chunk	capacity	mounted at
dev	*kbytes*	*used*	*avail*	*avg.chunk*	*capacity*	*mounted at*
dev	*kbytes*	*used*	*avail*	*avg.chunk*	*capacity*	*mounted at*
...	...	...	...	...	...	...

Dev is: the raw device name.

kbytes is: the capacity of the file system in kilobytes.

used is: the kilobytes used on the file system.

avail is: the kilobytes available (free).

avg.chunk is: the average size of free chunks.

capacity is: the percentage of the capacity which is used.

mounted at is: the place in the directory where the file system is mounted.

Sizes, free and used amounts are in kilo- (1024-) bytes.

If any filesystems are specified, only those file systems are listed. A file system may be given by either its directory name (eg: /auxfs) or its device name (eg: /dev/disc01).

-a shows the average size of free chunks (in kilo-bytes).

SEE ALSO

mount(1), disc(7)

FILES

/etc/mnttab /dev/discnn

NAME

diction, explain – print wordy sentences; thesaurus for diction

SYNTAX

diction [**–ml**] [**–m***x*] [**–n**] [**–f** pfile] file ...
explain
suggest

DESCRIPTION

Diction analyzes an unformatted text and puts brackets ([ ]) on the sentences that contain bad diction.

Before analyzing the text, *diction* removes all formatting commands by means of *deroff(1).*

–ml causes *deroff(1)* to skip lists; use this if the document contains many lists of non-sentences.

–f*pfile* specifies a user-supplied pattern file to be used in addition to the default file.

–n suppresses the default pattern file.

Explain and *suggest* are interactive thesauruses for the phrases found by diction.

SEE ALSO

deroff(1)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks.

NAME

diff – differential file comparator

SYNTAX

diff [– **efbh**] file1 file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is – , the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory with the name *file2* (*file1*) is used. The normal output contains lines of these forms:

n1 **a** *n3,n4*
n1,n2 **d** *n3*
n1,n2 **c** *n3,n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging **a** for **d** and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by <, then all the lines that are affected in the second file flagged by >.

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

The – **e** option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The – **f** option produces a similar script, not useful with *ed*, in the opposite order. In connection with – **e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A "latest version" appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option – **h** does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options – **e** and – **f** are unavailable with – **h**.

FILES

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

SEE ALSO

cmp(1), comm(1), ed(1).

DIAGNOSTICS

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

BUGS

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

NAME

diff3 – 3-way differential file comparison

SYNTAX

diff3 [– **ex3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

=====	all three files differ
=====1	*file1* is different
=====2	*file2* is different
=====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after the **c**. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the – **e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, i.e., the changes that normally would be flagged ===== and =====3. Option – **x** (– **3**) produces a script to incorporate only changes flagged ===== (=====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed – file1
```

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1).

BUGS

Text lines that consist of a single . will defeat – **e**.
Files longer than 64K bytes won't work.

NAME

dir - list floppy disc directory

SYNTAX

dir [-0] [-1]

DESCRIPTION

Dir lists the directory of a floppy disc on the standard output. The first line of the listing contains the volume name and date of creation, whether it is single or double sided, single or double density, and the number of 512-byte blocks. For each file in the directory, the name, number of blocks, creation date, starting block number, number of bytes in the last block, and the file type are listed. The last line shows the number of files and the number of used and unused blocks in the volume.

The default disc drive is floppy drive 0; the -0 option specifies the floppy drive 0, and -1 specifies floppy drive 1.

SEE ALSO

copyfloppy(1), rm(1), zero(1)

NOTES

An error message and number is printed if the disc is not mounted, unreadable, or if the drive is not functioning correctly.

NAME

dircmp – directory comparison

SYNTAX

dircmp [– **d**] [– **s**] [– **w***n*] dir1 dir2

DESCRIPTION

Dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

– **d** Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

– **s** Suppress messages about identical files.

– **w***n* Change the width of the output line to *n* characters. The default width is 72.

SEE ALSO

cmp(1), diff(1).

NAME

disptype – determine the type of Ridge display

SYNTAX

disptype [-l] [-s]

DESCRIPTION

Disptype is used to determine the type of the Ridge display *disp*(7). It is typically used in shell command files such as *.profile* or *.login* to help set terminal characteristics.

The file descriptor associated with the standard input is tested to determine the type of the display. With no arguments, a decimal number is printed on the standard output, representing the following display types:

0 monochrome, 768x1024, V1 keyboard

1 monochrome, 1024x800, V1 keyboard

2 monochrome, 1024x800, V2 keyboard

The options are interpreted as follows:

-l Print the long descriptive information instead of the number.

-s Silent; just return exit codes, printing nothing.

DIAGNOSTICS

Exit code 0 is returned if the standard input is a Ridge display; the exit code is 1 if it is not a Ridge display, and it is 2 for a bad argument.

SEE ALSO

Ridge Multi-Window Display Management Guide
windows(3), disp(7)

NAME

du – summarize disk usage

SYNTAX

du [– **ars**] [names]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each directory and file specified by the *names* argument. If *names* is missing, . is used. The block count is the number of 1024-byte blocks. The extra 4096-byte page taken by a multiple extent file is not included in the block count.

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

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

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

BUGS

If the – **a** option is not used, non-directories given as arguments are not listed.
If there are too many distinct linked files, *du* will count the excess files more than once.

NAME

dump – dump object file information

SYNTAX

dump *file*

DESCRIPTION

Dump displays information about the object file header, text segment, data segment, bss, symbol table, and file relocation. *File* must be a **.o** object file.

SEE ALSO

a.out(4)

NAME

echo – echo arguments

SYNTAX

echo [– **n**] [arg] ...

DESCRIPTION

Echo writes all *arg*s, separated by blanks, on the standard output. Normally, a newline character is output after the *arg*s.

– n specifies to omit the trailing newline character.

Echo also understands C-like escape conventions; beware of conflicts with the shell's use of ****:

\b	backspace
\c	print line without new-line
\f	form-feed
\n	new-line
\r	carriage return (but no newline)
\t	tab
****	backslash
*****n*	the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a zero.

Echo is useful for producing diagnostics in command files.

SEE ALSO

sh(1).

NAME

ed, red – text editor

SYNTAX

ed [–] [– **x**] [file]

red [–] [– **x**] [file]

DESCRIPTION

Ed is the standard text editor. If the *file* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited. The optional – suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the ! prompt after a *!shell command*. If – **x** is present, an *x* command is simulated first to handle an encrypted file. *Ed* operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

Red is a restricted version of *ed*. It will only allow editing of files in the current directory. It prohibits executing shell commands via *!shell command*. Attempts to bypass these restrictions result in an error message (*restricted shell*).

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

<:t5,10,15 s72:>

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

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, *no* commands are recognized; all input is merely collected. Input mode is left by typing a period (.) alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be *matched* by the RE. The REs allowed by *ed* are constructed as follows:

The following *one-character REs* match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets ([] ; see 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([])

(see 1.4 below).

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

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

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

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character *except* new-line and the remaining characters in the string. The ^ has this special meaning *only* if it occurs first in the string. The minus (-) may be used to indicate a range of consecutive ASCII characters; for example, **[0- 9]** is equivalent to **[0123456789]**. The - loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); e.g., **[]a- f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches *zero* or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; \{*m*\} matches *exactly m* occurrences; \{*m*,\} matches *at least m* occurrences; \{*m*,*n*\} matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an *entire RE* may be constrained to match only an initial segment or final segment of a line (or both):

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a *final* segment of a line.

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

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

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

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

1. The character . addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. *'x* addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A RE enclosed by slashes (/) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before *FILES* below.
6. A RE enclosed in question marks (?) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before *FILES* below.
7. An address followed by a plus sign (+) or a minus sign (−) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with + or − , the addition or subtraction is taken with respect to the current line; e.g, − 5 is understood to mean .− 5.
9. If an address ends with + or − , then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 immediately above, the address − refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to − .) Moreover, trailing + and − characters have a cumulative effect, so − − refers to the current line less 2.
10. For convenience, a comma (,) stands for the address pair 1,$, while a semicolon (;) stands for the pair .,$.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by a comma (,). They may also be separated by a semicolon (;). In the latter case, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. above). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are *not* part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **l**, **n** or **p**, in which case the current line is either listed, numbered or printed, respectively, as discussed below under the *l*, *n* and *p* commands.

(.)a

<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command: it causes the "appended" text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

(.,.)d

The *d*elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

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

E *file*

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

f *file*

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

(1,$)g/*RE***/***command list*

In the *g*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with . initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; *a*, *i*, and *c* commands and associated input are permitted; the . terminating input mode may be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the *p* command. The *g*, *G*, *v*, and *V* commands are *not* permitted in the *command list*. See also *BUGS* and the last paragraph before *FILES* below.

(1,$)G/*RE***/**

In the interactive *G*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, . is changed to that line, and any *one* command (other than one of the *a*, *c*, *i*, *g*, *G*, *v*, and *V* commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a new-line acts as a null command; an & causes the re-execution of the most recent command executed within the current invocation of *G*. Note that the commands input as part of the execution of the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

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

H

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

(.)i
<text>
.

The *i*nsert command inserts the given text before the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the *a* command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be a lower-case letter. The address *'x* then addresses this line; . is unchanged.

(.,.)l

The *l*ist command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., *tab*, *backspace*) are represented by (hopefully) mnemonic overstrikes, all other non-printing characters are printed in octal, and long lines are folded. An *l* command may be appended to any other command other than *e*, *f*, *r*, or *w*.

(.,.)m*a*

The *m*ove command repositions the addressed line(s) after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file; it is an error if address *a* falls within the range of moved lines; . is left at the last line moved.

(.,.)n

The *n*umber command prints the addressed lines, preceding each line by its line number and a tab character; . is left at the last line printed. The *n* command may be appended to any other command other than *e*, *f*, *r*, or *w*.

(.,.)p

The *p*rint command prints the addressed lines; . is left at the last line printed. The *p* command may be appended to any other command other than *e*, *f*, *r*, or *w*; for example, *dp* deletes the current line and prints the new current line.

P

The editor will prompt with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.

q

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

Q

The editor exits without checking if changes have been made in the buffer since the last *w* command.

(\$)r *file*

The *r*ead command reads in the given file after the addressed line. If no file name is given, the currently-remembered file name, if any, is used (see *e* and *f* commands). The currently-remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. Address 0 is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; . is set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose output is to be read. For example, "\$r !ls" appends current directory to the end of the file being edited. Such a shell command is *not* remembered as the current file name.

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

The *s*ubstitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) matched strings are replaced by the *replacement* if the global replacement indicator **g** appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or new-line may be used instead of / to delimit the RE and the *replacement*; . is left at the last line on which a substitution occurred. See also the last paragraph before *FILES* below.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of & in this context may be suppressed by preceding it by \. As a more general feature, the characters *n*, where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression of the specified RE enclosed between \(and \). When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

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

(.,.)t*a*

This command acts just like the *m* command, except that a *copy* of the addressed lines is placed after address *a* (which may be 0); . is left at the last line of the copy.

u

The *u*ndo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent *a*, *c*, *d*, *g*, *i*, *j*, *m*, *r*, *s*, *t*, *v*, *G*, or *V* command.

(1,\$)v/*RE*/*command list*

This command is the same as the global command *g* except that the *command list* is executed with . initially set to every line that does *not* match the RE.

(1,\$)V/*RE*/

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do *not* match the RE.

(1,\$)w *file*

The *w*rite command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your

umask setting (see *sh*(1)) dictates otherwise. The currently-remembered file name is *not* changed unless *file* is the very first file name mentioned since *ed* was invoked. If no file name is given, the currently-remembered file name, if any, is used (see *e* and *f* commands); . is unchanged. If the command is successful, the number of characters written is typed. If *file* is replaced by !, the rest of the line is taken to be a shell (*sh*(1)) command whose standard input is the addressed lines. Such a shell command is *not* remembered as the current file name.

X

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

(.)=

The line number of the addressed line is typed; . is unchanged by this command.

!*shell command*

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

(.+1)<new-line>

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

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

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name, and 128K characters in the buffer. The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

When reading a file, *ed* discards ASCII NUL characters and all characters after the last new-line. Files (e.g., **a.out**) that contain characters not in the ASCII set (bit 8 on) cannot be edited by *ed*.

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

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

FILES

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

DIAGNOSTICS

?	for command errors.
? *file*	for an inaccessible file.
	(use the *h*elp and *H*elp commands for detailed explanations).

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy *ed*'s buffer via the *e* or *q* commands: it prints **?** and allows one to continue editing. A second *e* or *q* command at this point will take effect. The – command-line option inhibits this feature.

SEE ALSO

crypt(1), grep(1), sed(1), sh(1), stty(1), fspec(4), regexp(5).

Tutorial - Text Editor in *Interactive Access to ROS,* and *Advanced Editing* in *ROS Document Processing Guide.*

CAVEATS AND BUGS

A *!* command cannot be subject to a *g* or a *v* command.
The *!* command and the ! escape from the *e*, *r*, and *w* commands cannot be used if the the editor is invoked from a restricted shell (see *sh*(1)).
The sequence \n in a RE does not match a new-line character.
The *l* command mishandles DEL.
Files encrypted directly with the *crypt*(1) command with the null key cannot be edited.
Characters are masked to 7 bits on input.

NAME

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

SYNTAX

edit [**– r**] name ...

DESCRIPTION

Edit is a variant of the text editor *ex* recommended for new or casual users who wish to use a command oriented editor. The following brief introduction should help you get started with *edit*. See *ex*(1) for other useful documents; in particular, if you are using a CRT terminal you will want to learn about the display editor *vi*.

BRIEF INTRODUCTION

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

Edit prompts for commands with the character ':', which you should see after starting the editor. If you are editing an existing file, then you will have some lines in *edit's* buffer (its name for the copy of the file you are editing). Most *edit* commands perform their actions on the "current line" unless otherwise specified. So, if you enter **print** (or abbreviated **p**) and press RETURN, this current line will be printed. If you **delete** (**d**) the current line, *edit* will print the new current line. When you start editing, *edit* makes the last line of the file the current line. If you **delete** this last line, then the new last line becomes the current one. In general, after a **delete,** the next line in the file becomes the current line. (Deleting the last line is a special case.)

If you start with an empty file, or wish to add some new lines, then the **append** (**a**) command can be used. After you enter this command (and press RETURN after the word **append**) *edit* reads lines from the terminal until you give a line consisting of just a ".", placing these lines after the current line. The last line you type then becomes the current line. The command **insert** (**i**) is like **append** but places the lines you give before, rather than after, the current line.

Edit numbers the lines in the buffer, with the first line having number 1. If you give the command "1" then *edit* will type this first line. If you then give the command **delete** *edit* will delete the first line, and line 2 will become line 1, and *edit* will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the **substitute** (**s**) command. You say "s/*old*/*new*/" where *old* is replaced by the old characters you want to get rid of and *new* is the new characters you want to replace it with.

The command **file** (**f**) will tell you how many lines there are in the buffer you are editing and will say "[Modified]" if you have changed it. After modifying a file you can put the buffer text back to replace the file by giving a **write** (**w**) command. You can then leave the editor by issuing a **quit** (**q**) command. If you run *edit* on a file, but don't change it, it is not necessary (but does no harm) to **write** the file back. If you try to **quit** from *edit* after modifying the buffer without writing it out, you will be warned that there has been "No **write** since last change" and *edit* will await another command. If you wish not to **write** the buffer out then you can issue another **quit** command. The buffer is then irretrievably discarded, and you return to the shell.

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

The **change** (**c**) command will change the current line to a sequence of lines you supply (as in **append** you give lines up to a line consisting of only a "."). You can tell **change** to change

more than one line by giving the line numbers of the lines you want to change, i.e. "3,5change". You can print lines this way too. Thus "1,23p" prints the first 23 lines of the file.

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

To look at the next line in the buffer, just press RETURN. To look at a number of lines, press ^D (control key and **D**) rather than RETIRN. This will show you a half screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at the text around where you are by giving the command "z.". The current line will then be the last line printed; you can get back to the line where you were before the "z." command by saying "^". The **z** command can also be given other following characters "z– " prints a screen of text (or 24 lines) ending where you are; "z+ " prints the next screenful. If you want less than a screenful of lines do, e.g., "z.12" to get 12 lines total. This method of giving counts works in general; thus you can delete 5 lines starting with the current line with the command "delete 5".

To find things in the file you can use line numbers if you happen to know them; since the line numbers change when you insert and delete lines this is somewhat unreliable. You can search backwards and forwards in the file for strings by giving commands of the form **/text/** to search forward for *text* or **?text?** to search backward for *text*. If a search reaches the end of the file without finding the text it wraps, end around, and continues to search back to the line where you are. A useful feature here is a search of the form **/^text/** which searches for *text* at the beginning of a line. Similarly **/text$/** searches for *text* at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has a symbolic name "."; this is most useful in a range of lines as in ".,$print" which prints the rest of the lines in the file. To get to the last line in the file you can refer to it by its symbolic name "$". Thus the command "$ delete" or "$d" deletes the last line in the file, no matter which line was the current line before. Arithmetic with line references is also possible. Thus the line "$– 5" is the fifth before the last, and ".+ 20" is 20 lines after the present.

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

SEE ALSO

ex(1), vi(1),

BUGS

See ex(1).

NAME

efl – Extended Fortran Language

SYNTAX

efl [options] [files]

DESCRIPTION

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

statement grouping with braces.

decision-making:

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

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

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

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

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

EFL also provides some syntactic "sugar":

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

comments:
this is a comment.

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

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

defines:
define *name replacement*

includes:
include *file*

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

An argument with an embedded = (equal sign) sets an EFL option as if it had appeared in an **option** statement at the start of the program. Many options are described in the reference manual. A set of defaults for a particular target machine may be selected by one of the choices: **system=unix**, **system=gcos**, or **system=cray**. The default setting of the **system** option is the same as the machine the compiler is running on.

Other specific options determine the style of input/output, error handling, continuation conventions, the number of characters packed per word, and default formats.

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

SEE ALSO

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

NAME

env – set environment for command execution

SYNTAX

env [–] [name=value] ... [command args]

DESCRIPTION

Env obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name=value* are merged into the inherited environment before the command is executed. The – flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

SEE ALSO

sh(1), exec(2), profile(4), environ(5).

NAME

error – analyze and disperse compiler error messages

SYNTAX

error [– **n**] [– **s**] [– **q**] [– **v**] [– **t** suffixlist] [– **I** ignorefile] [name]

DESCRIPTION

Error analyzes and optionally disperses the diagnostic error messages produced by a number of compilers and language processors to the source file and line where the errors occurred. It can replace the painful, traditional methods of scribbling abbreviations of errors on paper, and permits error messages and source code to be viewed simultaneously without machinations of multiple windows in a screen editor.

Error looks at the error messages, either from the specified file *name* or from the standard input, and attempts to determine which language processor produced each error message, determines the source file and line number to which the error message refers, determines if the error message is to be ignored or not, and inserts the (possibly slightly modified) error message into the source file as a comment on the line preceding to which the line the error message refers. Error messages which can't be categorized by language processor or content are not inserted into any file, but are sent to the standard output. *Error* touches source files only after all input has been read. By specifying the – **q** query option, the user is asked to confirm any potentially dangerous (such as touching a file) or verbose action. Otherwise *error* proceeds on its merry business. If the – **t** touch option and associated suffix list is given, *error* will restrict itself to touch only those files with suffices in the suffix list. Error also can be asked (by specifying – **v**) to invoke *vi*(1) on the files in which error messages were inserted; this obviates the need to remember the names of the files with errors.

Error is intended to be run with its standard input connected via a pipe to the error message source. Some language processors put error messages on their standard error file; others put their messages on the standard output. Hence, both error sources should be piped together into *error*. For example, when using the *csh* syntax,

make – s lint |& error – q – v

will analyze all the error messages produced by whatever programs *make* runs when making lint.

Error knows about the error messages produced by: *make, cc, cpp, ccom, as, ld, lint,* and *f77*. *Error* knows a standard format for error messages produced by the language processors, so is sensitive to changes in these formats. For all languages, error messages are restricted to be on one line. Some error messages refer to more than one line in more than one file; *error* will duplicate the error message and insert it at all of the places referenced.

Error will do one of six things with error messages.

synchronize

Some language processors produce short errors describing which file it is processing. *Error* uses these to determine the file name for languages that don't include the file name in each error message. These synchronization messages are consumed entirely by *error*.

discard Error messages from *lint* that refer to one of the two *lint* libraries, */usr/lib/llib-lc* and */usr/lib/llib-port* are discarded, to prevent accidently touching these libraries. Again, these error messages are consumed entirely by *error*.

nullify Error messages from *lint* can be nullified if they refer to a specific function, which is known to generate diagnostics which are not interesting. Nullified error messages are not inserted into the source file, but are written to the standard output. The names of functions to ignore are taken from either the file named *.errorrc* in the

users's home directory, or from the file named by the – I option. If the file does not exist, no error messages are nullified. If the file does exist, there must be one function name per line.

not file specific
Error messages that can't be intuited are grouped together, and written to the standard output before any files are touched. They will not be inserted into any source file.

file specific Error message that refer to a specific file, but to no specific line, are written to the standard output when that file is touched.

true errors Error messages that can be intuited are candidates for insertion into the file to which they refer.

Only true error messages are candidates for inserting into the file they refer to. Other error messages are consumed entirely by *error* or are written to the standard output. *Error* inserts the error messages into the source file on the line preceding the line the language processor found in error. Each error message is turned into a one line comment for the language, and is internally flagged with the string "*###*" at the beginning of the error, and "*%%%*" at the end of the error. This makes pattern searching for errors easier with an editor, and allows the messages to be easily removed. In addition, each error message contains the source line number for the line the message refers to. A reasonably formatted source program can be recompiled with the error messages still in it, without having the error messages themselves cause future errors. For poorly formatted source programs in free format languages, such as C or Pascal, it is possible to insert a comment into another comment, which can wreak havoc with a future compilation. To avoid this, programs with comments and source on the same line should be formatted so that language statements appear before comments.

Options available with *error* are:

– **n** Do *not* touch any files; all error messages are sent to the standard output.

– **q** The user is *queried* whether s/he wants to touch the file. A "y" or "n" to the question is necessary to continue. Absence of the – **q** option implies that all referenced files (except those referring to discarded error messages) are to be touched.

– **v** After all files have been touched, overlay the visual editor *vi* with it set up to edit all files touched, and positioned in the first touched file at the first error. If *vi* can't be found, try *ex* or *ed* from standard places.

– **t** Take the following argument as a suffix list. Files whose suffixes do not appear in the suffix list are not touched. The suffix list is dot separated, and "*" wildcards work. Thus the suffix list:

".c.y.foo*.h"

allows *error* to touch files ending with ".c", ".y", ".foo*" and ".h".

– **s** Print out *statistics* regarding the error categorization. Not too useful.

Error catches interrupt and terminate signals, and if in the insertion phase, will orderly terminate what it is doing.

FILES

~/.errorrc	function names to ignore for *lint* error messages
/dev/tty	user's teletype

BUGS

Opens the teletype directly to do user querying.

Source files with links make a new copy of the file with only one link to it.

Changing a language processor's format of error messages may cause *error* to not understand the error message.

Error, since it is purely mechanical, will not filter out subsequent errors caused by 'floodgating' initiated by one syntactically trivial error. Humans are still much better at discarding these related errors.

Error was designed for work on CRT's at reasonably high speed. It is less pleasant on slow speed terminals, and has never been used on hardcopy terminals.

NAME

ex – text editor

SYNTAX

ex [–] [– **v**] [– **t** tag] [– **r**] [+*command*] [– **l**] name ...

DESCRIPTION

Ex is the root of a family of editors: *edit, ex* and *vi. Ex* is a superset of *ed,* with the most notable extension being a display editing facility. Display based editing is the focus of *vi.*

If you have not used *ed,* or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed.*

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(1), which is a command which focuses on the display editing portion of *ex.*

DOCUMENTATION

The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers.

The *Ex Reference Manual – Version 3.1* is a comprehensive and complete manual for the command mode features of *ex,* but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex* see the editing documents written by Brian Kernighan for the editor *ed;* the material in the introductory and advanced documents works also with *ex.*

An Introduction to Display Editing with Vi introduces the display editor *vi* and provides reference material on *vi.* The *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction.*

FOR ED USERS

If you have used *ed* you will find that *ex* has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi.* Generally, the editor uses far more of the capabilities of terminals than *ed* does, and uses the terminal capability data base *termcap*(5) and the type of the terminal you are using from the variable TERM in the environment to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its **visual** command (which can be abbreviated **vi**) and which is the central mode of editing when using *vi*(1). There is also an interline editing **open** (**o**) command which works on all terminals.

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Press ^D to cause the editor to scroll a half-window of text; this is more useful than RETURN for stepping through a file quickly. Of course, the screen oriented **visual** mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. *Ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files unless you edited them so that you don't accidentally clobber with a *write* a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the editor **recover** command to retrieve your work. This will get you back to within a few lines of where you left off.

Ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (**n**) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, filenames in the editor may be formed with full

shell metasyntax. The metacharacter '% is also available in forming filenames and is replaced by the name of the current file.

For moving text between files and within a file the editor has a group of buffers, named *a* through *z*. You can place text in these named buffers and carry it over when you edit another file.

There is a command **&** in *ex* which repeats the last **substitute** command. In addition there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

You can use the **substitute** command in *ex* to systematically convert the case of letters between upper and lower case. It is possible to ignore case of letters in searches and substitutions. *Ex* also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word "edit" if your document also contains the word "editor."

Ex has a set of *options* which you can set to tailor it to your liking. One option which is very useful is the *autoindent* option which allows the editor to automatically supply leading white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new useful features include an intelligent **join (j)** command which supplies white space between joined lines automatically, commands < and > which shift groups of lines, and the ability to filter portions of the buffer through commands such as *sort*.

FILES

/usr/lib/ex?.?recover	recover command
/usr/lib/ex?.?preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), edit(1), grep(1), termcap(5), vi(1)

AUTHOR

University of California, Berkeley.

BUGS

The *undo* command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The *z* command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line '– ' option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

expand, unexpand - expand tabs to spaces, and vice versa

SYNTAX

expand [-tabstop] [-tab1,tab2,...tabn] [file ...]

DESCRIPTION

Expand converts the tab characters in the named files or standard input into blank characters. Backspace characters are preserved in the output, and decrement the column count for tab calculations. *Expand* is useful in creating character files from files with tabs in preparation for sorting, column seraching, etc.

If a single *tabstop* argument is given, then tabs are set *tabstop* spaces apart instead of the default 8. If multiple tabstops are given then the tabs are set at those specific columns.

Unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default only leading blanks and tabs are reconverted to maximal strings of tabs. If the **-a** option is given, then tabs are inserted wherever they would compress the resultant file by replacing two or more characters.

NAME

expr – evaluate arguments as an expression

SYNTAX

expr arguments

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to the shell must be escaped. Note that **0** is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2's complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within {} symbols.

expr \| *expr*
returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr \& *expr*
returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr { =, \>, \>=, \<, \<=, != } *expr*
returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr { +, - } *expr*
addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
multiplication, division, or remainder of the integer-valued arguments.

expr : *expr*
The matching operator : compares the first argument with the second argument which must be a regular expression; regular expression syntax is the same as that of *ed*(1), except that all patterns are "anchored" (i.e., begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (**0** on failure). Alternatively, the \(... \) pattern symbols can be used to return a portion of the first argument.

EXAMPLES

1. a=`expr $a + 1`

   adds 1 to the shell variable **a**.

2. # For $a equal to either "/usr/abc/file" or just "file"
   expr $a : '.*/\(.*\)' \| $a

   returns the last segment of a path name (i.e., file). Watch out for / alone as an argument: *expr* will take it as the division operator (see BUGS below).

3. # A better representation of example 2.
   expr //$a : '.*/\(.*\)'

   The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. expr $VAR : '.*'

   returns the number of characters in $VAR.

SEE ALSO

ed(1), sh(1).

EXIT CODE

As a side effect of expression evaluation, *expr* returns the following exit values:

0	if the expression is neither null nor **0**
1	if the expression *is* null or **0**
2	for invalid expressions.

DIAGNOSTICS

syntax error	for operator/operand errors
non-numeric argument	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If **$a** is an =, the command:

```
expr $a = =
```

looks like:

```
expr = = =
```

as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:

```
expr X$a = X=
```

NAME

f77 – FORTRAN 77 compiler

SYNTAX

F77 [*options*] ... *file* ...

DESCRIPTION

F77 activates the FORTRAN compiler.

File arguments ending with **.f** are taken to be FORTRAN 77 source programs. A compiled object file gets the same name as the source, but with **.o** substituted for the **.f** .

File arguments ending with **.F** are preprocessed by the C preprocessor before being compiled by *F77*.

File arguments ending with **.e** or **.r** are taken to be EFL or Ratfor source programs, respectively. These are preprocessed by the appropriate preprocessor before FORTRAN compilation.

File arguments ending with **.c** or **.s** are take to be C or assembly source programs, respectively, and are compiled or assembled appropriately, producing a **.o** file.

See *ld(1)* for link-time options.

–66 Compile FORTRAN 66 code only. Use of FORTRAN 77 features and extensions will produce fatal errors if this option is used.

–c Suppress loading and produce **.o** files for each source file.

–d*nnn (,nnn)*
Turns on selected debugging output. *nnn* turns different statements on and must be in the range of 0 - 20.

–g Produce additional symbol table information.

–o*output*
Name the final output file *output* instead of **a.out**.

–w Suppress all warning messages.

–w66 Suppress FORTRAN 66-compatible warnings only.

–D*name=def*
Define *name* to the C preprocessor, as if by "#define". If no "*=def*" is given, the *name* is defined as "1". (For **.F** files only).

–I*dir* Search for "#include" files (whose names do not start with "/") in the directory of *file*, then search in directory *dir*, then in directories in a standard list.

–O Invoke the global and "peephole" optimizers. See **BUGS**.

–O1 Invoke the global optimizer. Use on compilable programs only. See **BUGS**.

–O2 Invoke the post-pass peephole optimizer.

–S Compile the named *files*, and leave the assembly-language output in files with **.s** suffixes. Do not create **.o** files.

–T*x name*
Use an alternate pass or alternate temporary prefix. *x* can be one of the following letters:

1 Compiler

2	Pass2
a	Assembler
l	Loader
F	Start module (crt0.o)
m	Macro preprocessor (m4)
t	Temporary file prefix (fort)

name is the full path name of the alternate pass or the alternate temporary prefix.

–i2 Define undeclared integer constants and variables as 16 bits. (32 bits is the default size of undeclared integers.)

–i4 Use the default 32-bit size integers.

–l*x* Use the paragraph describing the -l*x* option to *ld(1)*.

–m4 Use the *M4(1)* preprocessor on each **.r** file before using Ratfor or EFL.

–map produce a loadmap in the file *file*.**map**

–onetrip
or

–1 Compile DO-loops to be executed at least once if reached. (Standard F77 DO-loops are not executed at all if the iteration test fails initially.)

–u Set the type of undefined variables to "undefined" rather than use FORTRAN rules for defining undefined variables.

–v Print on stderr the string passed to *exec(2)* just before each of the passes is invoked.

–C Compile code that will check that subscripts are within the declared array bounds.

–F Use the Ratfor or C preprocessors on appropriate files, and put the result if a file with a **.f** suffix, but do not compile it.

–Ex Use the string *x* as an EFL option in processing **.e** files.

–Rx Use the string *x* as a Ratfor option in processing **.r** files.

–M Produce a primitive load map, listing the names of the files which will be loaded.

–N[qxscn]*nnn*
Make static tables in the compiler bigger. The compiler will issue a message if this will be necessary.

q	Maximum number of equivalenced tables. Default is 800.
x	Maximum number of external names (common block names, subroutine, and function names). Default is 300.
s	Maximum number of statement numbers. Default is 401.
c	Maximum depth of nesting for control statements (e.g. DO loops). Default is 20.
n	Maximum hash table size. Default is 1009.

–p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if link editing takes place, replace the standard startoff routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

–U Do not convert uppercase letters to lowercase. The default is to convert FORTRAN programs to lowercase except within character string constants.

–Z*aaa* Pass the letters *aaa* directly to pass2 of the compiler.

Other arguments are taken to be loader option arguments, or F77 object programs typically produced by a previous run, or names of F77 routine libraries. These programs, together with the results of compilations specified, are loaded (in the order given) to produce an executable program file named **a.out**.

FILES

file.[fFresc]	input file
file.o	object file
a.out	loaded output
/usr/lib/f77pass1	compiler
/lib/f1	pass 2
/lib/c2	optional optimizer
/lib/cpp	C preprocessor
/usr/lib/libF77.a	intrinsic function library
/usr/lib/libI77.a	FORTRAN I/O library
/lib/libc.a	C library; see section (3)

SEE ALSO

cc(1), ld(1), efl(1), ratfor(1)

BUGS

The optimizer occasionally makes mistakes. Avoid it when debugging if obviously incorrect results are obtained.

NAME

factor – factor a number

SYNTAX

factor [number]

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2×10^{16}) it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If *factor* is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime.

DIAGNOSTICS

"Ouch" for input out of range or for garbage input.

NAME

file – determine file type

SYNTAX

file [– **c**] [– **f** ffile] [– **m** mfile] arg ...

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable **a.out**, *file* will print the version stamp, provided it is greater than 0 (see *ld*(1)).

If the – **f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

File uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of **/etc/magic** explains its format.

The – **m** option instructs *file* to use an alternate magic file.

The – **c** flag causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under – **c**.

SEE ALSO

ld(1)

NAME

find – find files

SYNTAX

find path-name-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each path name in the *path-name-list* (i.e., one or more path names) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, –*n* means less than *n* and *n* means exactly *n*.

–name *file* True if *file* matches the current file name. Normal shell argument syntax may be used if escaped (watch out for [, ? and *).

–perm *onum* True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(2)) become significant and the flags are compared:

(flags&onum)==onum

–type *c* True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, or plain file.

–links *n* True if the file has *n* links.

–user *uname* True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the **/etc/passwd** file, it is taken as a user ID.

–group *gname* True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the **/etc/group** file, it is taken as a group ID.

–size *n* True if the file is *n* blocks long (1024 bytes per block). Block count = (((actual size in bytes)+4095)/4096)*4

–atime *n* True if the file has been accessed in *n* days.

–mtime *n* True if the file has been modified in *n* days.

–ctime *n* True if a file has been changed in *n* days.

–exec *cmd* True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument **{}** is replaced by the current path name.

–ok *cmd* Like **–exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.

–print Always true; causes the current path name to be printed.

–cpio *device* Write the current file on *device* in *cpio*(4) format (5120 byte records).

–newer *file* True if the current file has been modified more recently than the argument *file*.

(*expression* **)** True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

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

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

3) Alternation of primaries (**-o** is the *or* operator).

EXAMPLE

To obtain the pathname of file **dog** whose location is not known:

find / -name dog -print

To find all files ending with **.e**:

find / -name '*.e' -print

To remove all files named **a.out** or ***.o** that have not been accessed for 7 days:

find / \(-name a.out -o -name '*.o' \) -atime +7 -exec rm {} \;

FILES

/etc/passwd, /etc/group

SEE ALSO

cpio(1), sh(1), test(1), stat(2), cpio(4).

NAME

fmt – simple text formatter

SYNTAX

fmt [name ...]

DESCRIPTION

Fmt is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces, on standard output, a version of its input with lines as close to 72 characters long as possible. The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing.

Fmt is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the *ex* editor (e.g. *vi*) the command

```
!}fmt
```

will reformat a paragraph, evening the lines.

SEE ALSO

nroff(1), mail(1)

AUTHOR

University of California, Berkeley

BUGS

The program was designed to be simple and fast – for more complex operations, the standard text processors are likely to be more appropriate.

NAME

fold - fold long lines for finite width output device

SYNTAX

fold [-width] [file ...]

DESCRIPTION

Fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using *expand*(1) before coming to *fold*.

SEE ALSO

expand(1)

BUGS

If underlining is present, it may be altered by folding.

NAME

fpr – print Fortran file

SYNTAX

fpr

DESCRIPTION

Fpr is a filter that transforms files formatted according to Fortran's carriage control conventions into files formatted according to UNIX line printer conventions.

Fpr copies its input onto its output, replacing the carriage control characters with characters that will produce the intended effects when printed using *lpr*(1). The first character of each line determines the vertical spacing as follows:

Character	*Vertical Space Before Printing*
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

A blank line is treated as if its first character is a blank. A blank that appears as a carriage control character is deleted. A zero is changed to a newline. A one is changed to a form feed. The effects of a "+ " are simulated using backspaces.

EXAMPLES

a.out |fpr |lpr

fpr < f77.output |lpr

BUGS

Results are undefined for input lines longer than 170 characters.

NAME

from – who is my mail from?

SYNTAX

from [– **s** sender] [user]

DESCRIPTION

From prints out the mail header lines in your mailbox file to show you who your mail is from. If *user* is specified, then *user*'s mailbox is examined instead of your own. If the -s option is given, then only headers for mail sent by *sender* are printed.

FILES

/usr/spool/mail/*

SEE ALSO

mail(1), prmail(1)

NAME

fsplit – split FORTRAN or ratfor files

SYNTAX

fsplit options files

DESCRIPTION

Fsplit splits the named *file(s)* into separate files, with one procedure per file. A procedure includes *blockdata, function, main, program,* and *subroutine* program segments. Procedure *X* is put in file *X*.**f**, *X*.**r**, or *X*.**e** depending on the language option chosen, with the following exceptions: *main* is put in the file *MAIN*.[**fr**] and unnamed *blockdata* segments in the files *blockdataN*.[**fr**] where *N* is a unique integer value for each file.

The following *options* pertain:

– **f** (default) Input files are *FORTRAN.*

– **r** Input files are *ratfor.*

– **s** Strip *FORTRAN* input lines to 72 or fewer characters with trailing blanks removed.

SEE ALSO

csplit(1), ratfor(1), split(1).

NOTES

FORTRAN programs are expected in lower case.

NAME

ftp – file transfer program

SYNTAX

ftp [**–v**] [**–d**] [**–i**] [**–n**] [**–g**] [**host**]

DESCRIPTION

Ftp is the user interface to the ARPANET standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

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

! Invoke a shell on the local machine.

append *local-file* [*remote-file*]
Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII. This is the default type.

bell Arrange that a bell be sounded after each file transfer command is completed.

binary Set the file transfer *type* to support binary image transfer.

bye Terminate the FTP session with the remote server and exit *ftp*.

cd *remote-directory*
Change the working directory on the remote machine to *remote-directory*.

close Terminate the FTP session with the remote server, and return to the command interpreter.

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

debug [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified, it is used to set the debugging level. When debugging is on, *ftp* prints each command sent to the remote machine, preceded by the string "-->".

dir [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory* and, optionally, placing the output in *local-file*. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, output comes to the terminal.

form *format*
Set the file transfer *form* to *format*. The default format is "file".

get *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the local file name is not specified, it is given the same name it has on the remote machine. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

hash Toggle hash-sign ("#") printing for each data block transferred. The size of a data block is 4096 bytes.

glob Toggle file name globbing. With file name globbing enabled, each local file or pathname is processed for *csh*(1) metacharacters. These characters include "*?[ ]~{}". Remote files specified in multiple item commands (e.g., *mput*) are globbed by the remote server. With globbing disabled, all files and pathnames are treated literally.

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

lcd [*directory*]
Change the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
Print an abbreviated listing of the contents of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If no local file is specified, the output is sent to the terminal.

mdelete *remote-files*
Delete the specified files on the remote machine. If globbing is enabled, the specification of remote files will first be expanded using *ls*.

mdir *remote-files local-file*
Obtain a directory listing of multiple files on the remote machine and place the result in *local-file*.

mget *remote-files*
Retrieve the specified files from the remote machine and place them in the current local directory. If globbing is enabled, the specification of remote files will first be expanding using *ls*.

mkdir *directory-name*
Make a directory on the remote machine.

mls *remote-files local-file*
Obtain an abbreviated listing of multiple files on the remote machine and place the result in *local-file*.

mode [*mode-name*]
Set the file transfer *mode* to *mode-name*. The default mode is "stream" mode.

mput *local-files*
Transfer multiple local files from the current local directory to the current working directory on the remote machine.

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

prompt
Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default), any *mget* or *mput* will transfer all files.

put *local-file* [*remote-file*]
Store a local file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for bye.

quote *arg1 arg2 ...*
The arguments specified are sent, verbatim, to the remote FTP server. A single FTP reply code is expected in return.

recv *remote-file* [*local-file*]
A synonym for get.

remotehelp [*command-name*]
Request help from the remote FTP server. If a *command-name* is specified, it is supplied to the server as well.

rename [*from*] [*to*]
Rename the file *from* on the remote machine, to the file *to*.

rmdir *directory-name*
Delete a directory on the remote machine.

send *local-file* [*remote-file*]
A synonym for put.

sendport
Toggle the use of PORT commands. By default, *ftp* will attempt to use a PORT command when establishing a connection for each data transfer. If the PORT command fails, *ftp* will use the default data port. When the PORT commands are disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which ignore PORT commands but, incorrectly, indicate they've been accepted.

status Show the current status of *ftp*.

struct [*struct-name*]
Set the file transfer *structure* to *struct-name*. By default "stream" structure is used.

tenex Set the file transfer type to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer *type* to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified and the server requires it, *ftp* will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user will be prompted for it. Unless *ftp* is invoked with "auto-login" disabled, this process is done automatically on initial connection to the FTP server.

verbose
Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. If verbose is on, statistics regarding the efficiency of the transfer will be reported when the file transfer completes. By default, verbose is on.

? [*command*]
A synonym for help.

Command arguments which have embedded spaces may be quoted with quote (") marks.

FILE NAMING CONVENTIONS

Files specified as arguments to *ftp* commands are processed according to the following rules.

1) If the file name "–" is specified, the **stdin** (for reading) or **stdout** (for writing) is used.

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

3) Failing the above checks, if "globbing" is enabled, local file names are expanded according to the rules described for the *glob* command in *csh*(1).

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters which may affect a file transfer. The *type* may be one of "ascii", "image" (binary), "ebcdic", and "local byte size" (for PDP-10's and PDP-20's mostly). *Ftp* supports the ascii and image types of file transfer.

Ftp supports only the default values for the remaining file transfer parameters: *mode*, *form*, and *struct*.

OPTIONS

Options may be specified at the command line, or to the command interpreter.

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

The **–n** option restrains *ftp* from attempting "auto-login" upon initial connection. If auto-login is enabled, *ftp* will check the *.netrc* file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* will use the login name on the local machine as the user identity on the remote machine, then prompt for a password and, optionally, an account with which to login.

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

The **–d** option enables debugging.

The **–g** option disables file name globbing.

BUGS

Many FTP server implementations do not support the experimental operations such as *pwd* (print working directory). Aborting a file transfer does not work right; if attempted, the local **ftp** will have to be killed by hand.

Pipes, like **dir | more**, do not work. See File Naming Convention number 2.

Tenex is not guaranteed to work.

Mdel, **mls**, and **mdir** are not implemented.

NAME

ftpd – DARPA Internet File Transfer Protocol server

SYNTAX

/etc/ftpd [– **d**] [– **l**] [– **t***timeout*]

DESCRIPTION

Ftpd is the DARPA Internet File Transfer Prototocol server process. The server uses the TCP protocol and listens at the port specified in the "ftp" service specification; see *services*(4).

If the – **d** option is specified, each socket created will have debugging turned on (SO_DEBUG). With debugging enabled, the system will trace all TCP packets sent and received on a socket.

If the – **l** option is specified, each ftp session is logged on the standard output. This allows a line of the form '/etc/ftpd -l > /tmp/ftplog" to be used to conveniently maintain a log of ftp sessions.

The ftp server will timeout an inactive session after 60 seconds. If the – **t** option is specified, the inactivity timeout period will be set to *timeout*.

The ftp server currently supports the following ftp requests; case is not distinguished.

Request	Description
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory ("ls -lg")
MODE	specify data transfer *mode*
NLST	give name list of files in directory ("ls")
NOOP	do nothing
PASS	specify password
PORT	specify data connection port
QUIT	terminate session
RETR	retrieve a file
RNFR	specify rename-from file name
RNTO	specify rename-to file name
STOR	store a file
STRU	specify data transfer *structure*
TYPE	specify data transfer *type*
USER	specify user name
XCUP	change to parent of current working directory
XCWD	change working directory
XMKD	make a directory
XPWD	print the current working directory
XRMD	remove a directory

The remaining ftp requests specified in Internet RFC 765 are recognized, but not implemented.

Ftpd interprets file names according to the "globbing" conventions used by *csh*(1). This allows users to utilize the metacharacters "*? [] {}~".

Ftpd authenticates users according to three rules.

1) The user name must be in the password data base, */etc/passwd*, and not have a null password. In this case a password must be provided by the client before any file operations may be performed.

2) The user name must not appear in the file */etc/ftpusers*.

3) If the user name is "anonymous" or "ftp", an anonymous ftp account must be present in the password file (user "ftp"). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host's name).

SEE ALSO

ftp(1)

BUGS

There is no support for aborting commands.

The anonymous account is inherently dangerous and should be avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user id of the logged-in user, reverting to the super-user only when binding addresses to sockets. The possible security holes have been scrutinized, but are incomplete.

NAME

erase, hardcopy, tekset, td – graphical device routines and filters

SYNTAX

erase
tekset
td [–**eurn**] [GPS~file~...]

DESCRIPTION

All of the commands described below reside in **/usr/bin/graf** (see *graphics*(1)).

erase *Erase* sends characters to a TEKTRONIX 4010 series storage terminal to erase the screen.

tekset *Tekset* sends characters to a TEKTRONIX terminal to clear the display screen, set the display mode to alpha, and set characters to the smallest font.

td *Td* translates a GPS to scope code for a TEKTRONIX 4010 series storage terminal. A viewing window is computed from the maximum and minimum points in *file* unless the –**u** or –**r** *option* is provided. If no *file* is given, the standard input is assumed. Options are:

- **e** Do not erase screen before initiating display.
- **r***n* Display GPS region *n*, *n* between 1 and 25 inclusive.
- **u** Display the entire GPS universe.

SEE ALSO

ged(1), graphics(1).
gps(4) in the *ROS Reference Manual.*

NAME

ged – graphical editor

SYNTAX

ged [–**euRrn**] [GPS~file~...]

DESCRIPTION

Ged is an interactive graphical editor used to display, construct, and edit GPS files on TEKTRONIX® 4010 series display terminals. If GPS *file*(s) are given, *ged* reads them into an internal display buffer and displays the buffer. The GPS in the buffer can then be edited. If – is given as a file name, *ged* reads a GPS from the standard input.

Ged accepts the following command line options:

e Do not erase the screen before the initial display.

r*n* Display region number *n*.

u Display the entire GPS *universe*.

A GPS file is composed of instances of three graphical objects: *lines*, *arc*, and *text*. *Arc* and *lines* objects have a start point, or *object-handle*, followed by zero or more points, or *point-handles*. *Text* has only an object-handle. The objects are positioned within a Cartesian plane, or *universe*, having 64K (–32K to +32K) points, or *universe-units*, on each axis. The universe is divided into 25 equal sized areas called *regions*. Regions are arranged in five rows of five squares each, numbered 1 to 25 from the lower left of the universe to the upper right.

Ged maps rectangular areas, called *windows*, from the universe onto the display screen. Windows allow the user to view pictures from different locations and at different magnifications. The *universe-window* is the window with minimum magnification, i.e., the window that views the entire universe. The *home-window* is the window that completely displays the contents of the display buffer.

COMMANDS

Ged commands are entered in *stages*. Typically each stage ends with a **<cr>** (return). Prior to the final **<cr>** the command may be aborted by typing **rubout**. The input of a stage may be edited during the stage using the erase and kill characters of the calling shell. The prompt * indicates that *ged* is waiting at stage 1.

Each command consists of a subset of the following stages:

1. *Command line*

   A *command line* consists of a command *name*, followed by *argument*(s). The command line is completed by a **<cr>**. A command *name* is a single character. Command *arguments* are either *option*(s) or a *file-name*. *Options* are indicated by a leading –.

2. *Text* *Text* is a sequence of characters terminated by an unescaped **<cr>**. (120 lines of text maximum.)

3. *Points* *Points* is a sequence of one or more screen locations (maximum of 30) indicated either by the terminal crosshairs or by name. The prompt for entering *points* is the appearance of the crosshairs. When the crosshairs are visible, typing:

   sp (space) enters the current location as a *point*. The *point* is identified with a number.

   $*n* enters the previous *point* numbered *n*.

   >*x* labels the last *point* entered with the upper case letter *x*.

$x enters the *point* labeled *x*.

. establishes the previous *points* as the current *points*. At the start of a command, the previous *points* are those locations given with the previous command.

= echoes the current *points*.

$.n enters the *point* numbered *n* from the previous *points*.

erases the last *point* entered.

@ erases all of the *points* entered.

4. *Pivot* The *pivot* is a single location, entered by typing **<cr>** or by using the **$** operator, and indicated with a *****.

5. *Destination*
The *destination* is a single location entered by typing **<cr>** or by using **$**.

COMMAND SUMMARY

In this summary, characters typed by the user are printed in **bold**. Command stages are printed in *italics*. Arguments surrounded by brackets "[]" are optional. Parentheses "()" surrounding arguments separated by "or" means that exactly one of the arguments must be given.

Construct commands:

Arc	[**—echo,style,weight**] ***points***
Box	[**—echo,style,weight**] ***points***
Circle	[**—echo,style,weight**] ***points***
Hardware	[**—echo**] ***text points***
Lines	[**—echo,style,weight**] ***points***
Text	[**—angle,echo,height,mid-point,right-point,text,weight**] ***text points***

Edit commands:

Delete	(**—** (**universe** or **view**) or ***points***)
Edit	[**—angle,echo,height,style,weight**] (**—** (**universe** or **view**) or ***points***)
Kopy	[**—echo,points,x**] ***points pivot destination***
Move	[**—echo,points,x**] ***points pivot destination***
Rotate	[**—angle,echo,kopy,x**] ***points pivot destination***
Scale	[**—echo,factor,kopy,x**] ***points pivot destination***

View commands:

coordinates	***points***
erase	
new-display	
object-handles	(**—** (**universe** or **view**) or ***points***)
point-handles	(**—** (**labelled-points** or **universe** or **view**) or ***points***)
view	(**—** (**home** or **universe** or **region**) or [**—x**] ***pivot destination***)

x	[**–view**] *points*
zoom	[**–out**] *points*

Other commands:

quit or Quit

read	[**–angle,echo,height,mid-point,right-point,text,weight**] *file-name* [*destination*]
set	[**–angle,echo,factor,height,kopy,mid-point,points, right-point,style,text,weight,x**]
write	*file-name*
!*command*	
?	

Options:

Options specify parameters used to construct, edit, and view graphical objects. If a parameter used by a command is not specified as an *option*, the default value for the parameter will be used (see set below). The format of command *options* is:

–option[,*option*]

where *option* is *keyletter*[*value*]. Flags take on the *values* of true or false, which are indicated by + and – respectively. If no *value* is given with a flag, true is assumed.

Object options:

angle*n*	Angle of *n* degrees.
echo	When true, echo additions to the display buffer.
factor*n*	Scale factor is *n* percent.
height*n*	Height of *text* is *n* universe-units ($0 \leq n < 1280$).
kopy	When true, copy rather than move.
mid-point	When true, mid-point is used to locate text string.
points	When true, operate on points. Otherwise, operate on objects.
right-point	When true, right-point is used to locate *text* string.
style*type*	Set line style to one of following *types*: **so** solid **da** dashed **dd** dot-dashed **do** dotted **ld** long-dashed
text	When false, *text* strings are outlined rather than drawn.
weight*type*	Sets line weight to one of following *types*: **n** narrow **m** medium **b** bold

Area options:

home	Reference the home-window.

out	Reduce magnification.
region*n*	Reference region *n*.
universe	Reference the universe-window.
view	Reference those objects currently in view.
x	Indicate the center of the referenced area.

COMMAND DESCRIPTIONS

Construct commands:

Arc and **Lines**

behave similarly. Each consists of a *command line* followed by *points*. The first *point* entered is the object-handle. Successive *points* are point-handles. **Lines** connect the handles in numerical order. **Arc** fits a curve to the handles (currently a maximum of 3 points will be fit with a circular arc; splines will be added in a later version).

Box and **Circle**

are special cases of **Lines** and **Arc**, respectively. **Box** generates a rectangle with sides parallel to the universe axes. A diagonal of the rectangle would connect the first *point* entered with the last *point*. The first *point* is the object-handle. Point-handles are created at each of the vertices. **Circle** generates a circular arc centered about the *point* numbered zero and passing through the last *point*. The circle's object-handle coincides with the last *point*. A point-handle is generated 180 degrees around the circle from the object-handle.

Text and **Hardware**

generate *text* objects. Each consists of a *command line*, *text* and *points*. *Text* is a sequence of characters delimited by **<cr>**. Multiple lines of text may be entered by preceding a **cr** with a backslash (i.e., **\cr**). The **Text** command creates software generated characters. Each line of software text is treated as a separate *text* object. The first *point* entered is the object-handle for the first line of text. The **Hardware** command sends the characters in *text* uninterpreted to the terminal.

Edit commands:

Edit commands operate on portions of the display buffer called *defined areas*. A defined area is referenced either with an area *option* or interactively. If an area *option* is not given, the perimeter of the defined area is indicated by *points*. If no *point* is entered, a small defined area is built around the location of the **<cr>**. This is useful to reference a single *point*. If only one *point* is entered, the location of the **<cr>** is taken in conjunction with the *point* to indicate a diagonal of a rectangle. A defined area referenced by *points* will be outlined with dotted lines.

Delete

removes all objects whose object-handle lies within a defined area. The **universe** option removes all objects and erases the screen.

Edit modifies the parameters of the objects within a defined area. Parameters that can be edited are:

angle	angle of *text*
height	height of *text*
style	style of *lines* and *arc*
weight	weight of *lines*, *arc*, and *text*.

Kopy (or **Move**)

copies (or moves) object- and/or point-handles within a defined area by the displacement from the *pivot* to the *destination*.

Rotate
rotates objects within a defined area around the *pivot*. If the kopy flag is true, then the objects are copied rather than moved.

Scale
For objects whose object handles are within a defined area, point displacements from the *pivot* are scaled by factor percent. If the kopy flag is true, then the objects are copied rather than moved.

View commands:

coordinates
prints the location of *point*(s) in universe- and screen-units.

erase
clears the screen (but not the display buffer).

new-display
erases the screen then displays the display buffer.

object-handles (or point-handles)
labels object-handles (and/or point-handles) that lie within the defined area with O (or P). Point-handles identifies labelled points when the labelled-points flag is true.

view moves the window so that the universe point corresponding to the *pivot* coincides with the screen point corresponding to the *destination*. Options for home, universe, and region display particular windows in the universe.

x indicates the center of a defined area. Option view indicates the center of the screen.

zoom
decreases (zoom out) or increases the magnification of the viewing window based on the defined area. For increased magnification, the window is set to circumscribe the defined area. For a decrease in magnification, the current window is inscribed within the defined area.

Other commands:

quit or Quit
exit from *ged*. Quit responds with ? if the display buffer has not been written since the last modification.

read inputs the contents of a file. If the file contains a GPS it is read directly. If the file contains text it is converted into *text* object(s). The first line of a text file begins at *destination*.

set when given *option*(s) resets default parameters, otherwise it prints current default values.

write
outputs the contents of the display buffer to a file.

! escapes *ged* to execute a UNIX system command.

? lists *ged* commands.

SEE ALSO

gdev(1), graphics(1), sh(1).
gps(4) in the *ROS Reference Manual*.

An Introduction to the Graphical Editor in the *UNIX System Graphics Guide*.

WARNING

See *ROS Utility Guide - Multi-Windows* section 7.2 for more information on *TEKTRONIX*® emulation.

NAME

get – get a version of an SCCS file

SYNTAX

get [– **r**SID] [– **c**cutoff] [– **i**list] [– **x**list] [– **a**seq-no.] [– **k**] [– **e**] [– l[p]] [– **p**] [– **m**] [– **n**] [– **s**] [– **b**] [– **g**] [– **t**] file ...

DESCRIPTION

Get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with – . The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also *FILES*, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

– **r***SID* The *S*CCS *ID*entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta*(1) if the – **e** keyletter is also used), as a function of the SID specified.

– **c***cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, – **c7502** is equivalent to – **c750228235959**. Any number of non-numeric characters may separate the various 2 digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: **"– c77/2/2 9:22:25"**. Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(1C) command:

~!get "– c%E% %U%" s.file

– **e** Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The – **e** keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *gets* for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file (see *admin*(1)). Concurrent use of **get** – **e** for different SIDs is always allowed.

If the *g-file* generated by *get* with an – **e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the – **k** keyletter in place of the – **e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see *admin*(1)) are enforced when the – **e** keyletter is used.

– **b** Used with the – **e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file (see *admin*(1)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)
Note: A branch *delta* may always be created from a non-leaf *delta*.

– i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID – SID

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

– x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the – **i** keyletter for the *list* format.

– **k** Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The – **k** keyletter is implied by the – **e** keyletter.

– **l**[**p**] Causes a delta summary to be written into an *l-file*. If – **lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

– **p** Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the – **s** keyletter is used, in which case it disappears.

– **s** Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

– **m** Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

– **n** Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the – **m** and – **n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the – **m** keyletter generated format.

– **g** Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

– **t** Used to access the most recently created ("top") delta in a given release (e.g., – **r1**), or release and level (e.g., – **r1.2**).

– a*seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter, and users should not use it. If both the – **r** and – **a** keyletters are specified, the – **a** keyletter is used. Care should be taken when using the – **a** keyletter in conjunction with the – **e** keyletter, as the SID of the delta to be created may not be what one expects. The – **r** keyletter can be used with the – **a** and – **e** keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the – **e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the – **i** keyletter is used included deltas are listed following the notation "Included";

if the – **x** keyletter is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	– **b** Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This forces creation of the *first* delta in a *new* release.

\# Successor.

† The – **b** keyletter is effective only if the **b** flag (see *admin*(1)) is present in the file. An entry of – means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file (see *admin*(1)), or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.

%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file (see *admin*(1)).
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file (see *admin*(1)).
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by *what*(1).
%W%	A shorthand notation for constructing *what*(1) strings for ROS program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(1) strings for non-ROS program files. %A% = %Z%%Y% %M% %I%%Z%

FILES

Several auxiliary files may be created by *get*, These files are known generically as the *g-file*, *l-file*, *p-file*, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the **s.** prefix. For example, **s.xyz.c**, the auxiliary file names would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **– p** keyletter is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. It is owned by the real user. If the **– k** keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the **– l** keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
   * otherwise.
b. A blank character if the delta was applied or wasn't applied and ignored;
   * if the delta wasn't applied and wasn't ignored.
c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included.
   "X": Excluded.
   "C": Cut off (by a **– c** keyletter).
d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.

h. Blank.
i. Login name of person who created *delta*.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* passes information resulting from a *get* with an – **e** keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an – **e** keyletter for the same SID until *delta* is executed or the joint edit flag, **j**, (see *admin*(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the – **i** keyletter argument if it was present, followed by a blank and the – **x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1), sccsfile(4).
Source Code Control System in the *ROS Utility Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the – **e** keyletter is used.

NAME

getopt – parse command options

SYNTAX

**set – – `getopt** optstring $*`

DESCRIPTION

Getopt breaks up options in command lines for easy parsing and option-checking by shell procedures. *Optstring* is a string of recognized option letters (see *getopt*(3C)); if a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. The special option – – delimits the end of the options. If it is used explicitly, *getopt* will recognize it; otherwise, *getopt* will generate it; in either case, *getopt* will place it at the end of the options. The shell's positional parameters ($1 $2 ...) are reset so that each option is preceded by a – and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an argument:

```
set – –  `getopt abo: $*`
if [ $? != 0 ]
then
        echo $USAGE
        exit 2
fi
for i in $*
do
        case $i in
        – a | – b)      FLAG=$i; shift;;
        – o)            OARG=$2; shift 2;;
        – – )           shift; break;;
        esac
done
```

This code will accept any of the following as equivalent:

```
cmd – aoarg file file
cmd – a – o arg file file
cmd – oarg – a file file
cmd – a – oarg – –  file file
```

SEE ALSO

sh(1), getopt(3C).

DIAGNOSTICS

Getopt prints an error message on the standard error when it encounters an option letter not included in *optstring*.

NAME

getty – set terminal characteristics

SYNTAX

/etc/getty *line* [*speed*]
/etc/getty – c *file*

DESCRIPTION

Getty is invoked by the User Monitor process during the system startup sequence. (The sequence is Startup, User Monitor, Getty, Login, Shell.) **Getty** reads instructions from the file **gettydefs(4)** on the line speed, accepts a user login name, determines the terminal characteristics from that name, and passes that name to the *login(1)* program.

Line is a tty device name contained in **/dev** which will become the control terminal.

/etc/gettydefs tells *getty* what speed to initially run at, what the login message should look like, what the inital tty settings are. **/etc/gettydefs** actually contains a list of specifications for *getty* to attempt, in order, until an appropriate specification is found.

Normally, *getty* reads the first line of **gettydefs** first, then reads successive lines until the information on a particular line is found to specify the correct speed. *Speed* (optional) tells *getty* which line of the **/etc/gettydefs** file to start reading speed information from.

By default, *getty* sets the *speed* of the interface to 9600 baud, specifies that raw mode is to be used (awaken on every character), that echo is to be suppressed, either parity allowed, that the connection is local, that newline characters will be converted to "CR-LF", and that tab expansion will be performed on the standard output.

Getty types the login message before reading the user's name a character at a time. If a null character (or framing error) is received, it is assumed to be the result of the user pushing the "break" key. This will cause *getty* to attempt the next *speed* entry in the **/etc/gettydefs** file.

The user's name is terminated by a new-line or carriage-return character. The "CR" results in the system being set to treat "CR's" appropriately (see *ioctl(2)*).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, *login* is called with the user's name as an argument. Additional arguments may be typed after the login name. These are passed to *login* which will place them in the environment (see *login (1)*).

– **c** *file* is a check option, in which **Getty** scans *file* instead of **/etc/gettydefs** and prints a diagnosis on the standard output. In this case, it prints the flag values for correct entries, and reports other improper constructions. See *ioctl (2)* to interpret the values. Note that some values are added to the flags automatically.

FILES

/etc/gettydefs

SEE ALSO

ct(1), login(1), ioctl(2), gettydefs(4), inittab(4), tty(7).

NAME

graph – draw a graph

SYNTAX

graph [options]

DESCRIPTION

Graph with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by the *tplot*(1) filters.

If the coordinates of a point are followed by a non-numeric string, that string is printed as a label beginning on the point. Labels may be surrounded with quotes ", in which case they may be empty or contain blanks and numbers; labels never contain new-lines.

The following options are recognized, each as a separate argument:

- **– a** Supply abscissas automatically (they are missing from the input); spacing is given by the next argument (default 1). A second optional argument is the starting point for automatic abscissas (default 0 or lower limit given by – **x**).
- **– b** Break (disconnect) the graph after each label in the input.
- **– c** Character string given by next argument is default label for each point.
- **– g** Next argument is grid style, 0 no grid, 1 frame with ticks, 2 full grid (default).
- **– l** Next argument is label for graph.
- **– m** Next argument is mode (style) of connecting lines: 0 disconnected, 1 connected (default). Some devices give distinguishable line styles for other small integers (e.g., the Tektronix 4014: 2=dotted, 3=dash-dot, 4=short-dash, 5=long-dash).
- **– s** Save screen, don't erase before plotting.
- **– x** [**l**] If **l** is present, x axis is logarithmic. Next 1 (or 2) arguments are lower (and upper) *x* limits. Third argument, if present, is grid spacing on *x* axis. Normally these quantities are determined automatically.
- **– y** [**l**] Similarly for *y*.
- **– h** Next argument is fraction of space for height.
- **– w** Similarly for width.
- **– r** Next argument is fraction of space to move right before plotting.
- **– u** Similarly to move up before plotting.
- **– t** Transpose horizontal and vertical axes. (Option – **x** now applies to the vertical axis.)

A legend indicating grid range is produced with a grid unless the – **s** option is present. If a specified lower limit exceeds the upper limit, the axis is reversed.

SEE ALSO

spline(1), tplot(1).

BUGS

Graph stores all points internally and drops those for which there isn't room.
Segments that run out of bounds are dropped, not windowed.
Logarithmic axes may not be reversed.

NAME

graphics – access graphical and numerical commands

SYNTAX

graphics

DESCRIPTION

Graphics prefixes the path name **/usr/bin/graf** to the current **$PATH** value, changes the primary shell prompt to ^, and executes a new shell. The directory **/usr/bin/graf** contains all of the Graphics subsystem commands. To restore the environment that existed prior to issuing the *graphics* command, type EOT (control-d on most terminals). To logoff from the graphics environment, type **quit**.

The command line format for a command in *graphics* is *command name* followed by *argument*(s). An *argument* may be a *file name* or an *option string*. A *file name* is the name of any UNIX system file except those beginning with –. The *file name* – is the name for the standard input. An *option string* consists of – followed by one or more *option*(s). An *option* consists of a keyletter possibly followed by a value. *Options* may be separated by commas.

The graphical commands have been partitioned into four groups.

Commands that manipulate and plot numerical data; see *stat*(1).

Commands that generate tables of contents; see *toc*(1).

Commands that interact with graphical devices; see *gdev*(1) and *ged*(1).

A collection of graphical utility commands; see *gutil*(1).

A list of the *graphics* commands can be generated by typing **whatis** in the *graphics* environment.

SEE ALSO

gdev(1), ged(1), gutil(1), stat(1), toc(1).
gps(4) in the *ROS Reference Manual.*

UNIX System Graphics Guide.

NAME

grep, egrep, fgrep – search a file for a pattern

SYNTAX

grep [options] expression [files]

egrep [options] [expression] [files]

fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular *expressions* in the style of *ed*(1); it uses a compact non-deterministic algorithm. *Egrep* patterns are full regular *expressions*; it uses a fast deterministic algorithm that sometimes needs exponential space. *Fgrep* patterns are fixed *strings*; it is fast and compact. The following *options* are recognized:

- **– v** All lines but those matching are printed.
- **– x** (Exact) only lines matched in their entirety are printed (*fgrep* only).
- **– c** Only a count of matching lines is printed.
- **– i** Ignore upper/lower case distinction during comparisons.
- **– l** Only the names of files with matching lines are listed (once), separated by new-lines.
- **– n** Each line is preceded by its relative line number in the file.
- **– b** Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context. (a block is 1024 bytes.)
- **– s** The error messages produced for nonexistent or unreadable files are suppressed (*grep* only).
- **– e** *expression*
  Same as a simple *expression* argument, but useful when the *expression* begins with a – (does not work with *grep*).
- **– f** *file*
  The regular *expression* (*egrep*) or *strings* list (*fgrep*) is taken from the *file*.

In all cases, the file name is output if there is more than one input file. Care should be taken when using the characters $, *, [, ^, |, (,), and \ in *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes '...'.

Fgrep searches for lines that contain one of the *strings* separated by new-lines.

Egrep accepts regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a new-line match strings that are matched by either.
4. A regular expression may be enclosed in parentheses ( ) for grouping.

The order of precedence of operators is [], then *? +, then concatenation, then | and new-line.

SEE ALSO

ed(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep*, but we do not know a single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to BUFSIZ characters; longer lines are truncated. (BUFSIZ is defined in **/usr/include/stdio.h**.)
Egrep does not recognize ranges, such as **[a– z]**, in character classes.
If there is a line with embedded nulls, *grep* will only match up to the first null; if it matches, it will print the entire line.

NAME

gutil – graphical utilities

SYNTAX

command-name [options] [files]

DESCRIPTION

Below is a list of miscellaneous device independent utility commands found in **/usr/bin/graf**. If no *files* are given, input is from the standard input. All output is to the standard output. Graphical data is stored in GPS format; see *gps*(4).

bel – send bel character to terminal

cvrtopt [=**s***string* **f***string* **i***string* **t***string*] [*args*] – options converter
Cvrtopt reformats *args* (usually the command line arguments of a calling shell procedure) to facilitate processing by shell procedures. An *arg* is either a file name (a string not beginning with a –, or a – by itself) or an option string (a string of options beginning with a –). Output is of the form:

–option –option . . . file name(s)

All options appear singularly and preceding any file names. Options that take values (e.g., –r1.1) or are two-letters long must be described through options to *cvrtopt*.

Cvrtopt is usually used with *set* in the following manner as the first line of a shell procedure:

set – `cvrtopt =**[*options*]** $@`

Options to *cvrtopt* are:

s*string* *String* accepts string values.

f*string* *String* accepts floating point numbers as values.

i*string* *String* accepts integers as values.

t*string* *String* is a two-letter option name that takes no value.

String is a one- or two-letter option name.

gd [GPS *files*] – GPS dump
Gd prints a human readable listing of GPS.

gtop [–**r***n* **u**] [GPS *files*] – GPS to *plot*(4) filter
Gtop transforms a GPS into *plot*(4) commands displayable by *plot* filters. Unless an *option* is given, GPS objects are translated if they fall within the window that circumscribes the first *file*.
Options:

r*n* translate objects in GPS region *n*.

u translate all objects in the GPS universe.

pd [*plot*(5) *files*] – *plot*(4) dump
Pd prints a human readable listing of *plot*(4) format graphical commands.

ptog [*plot*(5) *files*] – *plot*(4) to GPS filter
Ptog transforms *plot*(4) commands into a GPS.

quit – terminate session

remcom [*files*] – remove comments
Remcom copies its input to its output with comments removed. Comments are as defined in C (i.e., /* comment */).

whatis [−**o**] [*names*] − brief on-line documentation
Whatis prints a brief description of each *name* given. If no *name* is given, then the current list of description *names* is printed. The command **whatis *** prints out every description.
Option:

o just print command options

yoo *file* − pipe fitting
Yoo is a piping primitive that deposits the output of a pipeline into a *file* used in the pipeline. Without *yoo*, this operation is usually not successful, as it causes a read and write on the same file simultaneously.

SEE ALSO

graphics(1).
gps(4), plot(4) in the *ROS Reference Manual.*

NAME

head - give first few lines of a stream

SYNTAX

head [-count] [file ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1)

NAME

help – ask for help

SYNTAX

help [args]

DESCRIPTION

Help finds information to explain a message from an sccs command or explain the use of an sccs command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1 Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., **ge6**, for message 6 from the *get* command).

type 2 Does not contain numerics (as a command, such as **get**)

type 3 Is all numeric (e.g., **212**)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try "help stuck".

FILES

/usr/lib/help directory containing files of message text.

/usr/lib/help/helploc file containing locations of help files not in **/usr/lib/help**.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

hexdump – hexadecimal file dump

SYNTAX

hexdump [ ] ...

DESCRIPTION

Hexdump reads each input **file** in sequence and prints their contents on the standard output in hexadecimal notation. Each output line consists of a byte number within the file, followed by 16 hexadecimal bytes, and 16 equivalent ASCII characters from the input. Non-printable hexadecimal bytes are represented in the ASCII translation as ".".

If no input **file** is given, the standard input is read and dumped to the standard output.

NAME

hostid – set or print identifier of current host system

SYNTAX

hostid [identifier]

DESCRIPTION

The *hostid* command prints the identifier of the current host in hexadecimal. This numeric value is expected to be unique across all hosts and is normally set to the host's Internet address. The super-user can set the hostid by giving a hexadecimal argument; this is usually done in the startup script **/etc/rc**.

NAME

hostname – set or print name of current host system

SYNTAX

hostname [nameofhost]

DESCRIPTION

Hostname prints the name of the current host. The super-user can set the hostname by giving an argument; this is usually done in the startup script **/etc/rc**.

NAME

id – print user and group IDs and names

SYNTAX

id

DESCRIPTION

Id writes a message on the standard output giving the user and group IDs and the corresponding names of the invoking process. If the effective and real IDs do not match, both are printed.

SEE ALSO

logname(1),

NAME

ifconfig – configure network interface parameters

SYNTAX

/etc/ifconfig interface [*address*] [*parameters*]

DESCRIPTION

Ifconfig is used to assign an address to a network interface and/or configure network interface parameters. *Ifconfig* must be used at boot time to define the network address of each interface present on a machine; it may also be used at a later time to redefine an interface's address. The *interface* parameter is a string of the form "name unit", e.g. "ec0", while the address is either a host name present in the host name data base, *hosts*(5), or a DARPA Internet address expressed in the Internet standard "dot notation".

The following parameters may be set with *ifconfig*:

up Mark an interface "up".

down Mark an interface "down". When an interface is marked "down", the system will not attempt to transmit messages through that interface.

trailers Enable the use of a "trailer" link level encapsulation when sending. If a network interface supports *trailers*, the system will, when possible, encapsulate outgoing messages in a manner which minimizes the number of memory to memory copy operations performed by the receiver.

–trailers Disable the use of a "trailer" link level encapsulation (default).

arp Enable the use of the Address Resolution Protocol in mapping between network level addresses and link level addresses (default). This is currently implemented for mapping between DARPA Internet addresses and 10Mb/s Ethernet addresses.

–arp Disable the use of the Address Resolution Protocol.

Ifconfig displays the current configuration for a network interface when no optional parameters are supplied.

Only the super-user may modify the configuration of a network interface.

DIAGNOSTICS

Messages indicating the specified interface does not exist, the requested address is unknown, the user is not privileged and tried to alter an interface's configuration.

NAME

indent – indent and format C program source

SYNTAX

indent *input* [*output*] [*flags*]

DESCRIPTION

Indent is intended primarily as a C program formatter. Specifically, *indent* will:

- indent code lines
- align comments
- insert spaces around operators where necessary
- break up declaration lists as in "int a,b,c;".

Indent will not break up long statements to make them fit within the maximum line length, but it will flag lines that are too long. Lines will be broken so that each statement starts a new line, and braces will appear alone on a line. (See the –br option to inhibit this.) Also, an attempt is made to line up identifiers in declarations.

If the *flags* are specified, they may appear before or after the file names. If the *output* file is omitted, the formatted file will be written back into *input* and a "backup" copy of *input* will be written in the current directory. If *input* is named "/blah/blah/file", the backup file will be named ".Bfile". If *output* is specified, *indent* checks to make sure it is different from *input*.

The following flags may be used to control the formatting style imposed by *indent*.

–l*nnn* Maximum length of an output line. The default is 75.

–c*nnn* The column in which comments will start. The default is 33.

–cd*nnn* The column in which comments on declarations will start. The default is for these comments to start in the same column as other comments.

–i*nnn* The number of spaces for one indentation level. The default is 4.

–dj,–ndj **–dj** will cause declarations to be left justified. **–ndj** will cause them to be indented the same as code. The default is **–ndj**.

–v,–nv **–v** turns on "verbose" mode, **–nv** turns it off. When in verbose mode, *indent* will report when it splits one line of input into two or more lines of output, and it will give some size statistics at completion. The default is **–nv**.

–bc,–nbc If **–bc** is specified, then a newline will be forced after each comma in a declaration. **–nbc** will turn off this option. The default is **–bc**.

–d*nnn* This option controls the placement of comments, which are not to the right of code. Specifying **–d2** means that such comments will be placed two indentation levels to the left of code. The default **–d0** lines up these comments with the code. See the section on comment indentation below.

–br,–bl Specifying **–bl** will cause complex statements to be lined up like this:

```
if (...)
{
    code
}
```

Specifying **–br** (the default) will make them look like this:

```
if (...) {
    code
}
```

You may set up your own "profile" of defaults to *indent* by creating the file ".indent.pro" in your login directory and including whatever switches you like. If *indent* is run and a profile file exists, then it is read to set up the program's defaults. Switches on the command line, though, will always override profile switches. The profile file must be a single line of not more than 127 characters. The switches should be separated on the line by spaces or tabs.

Multi-line expressions

Indent will not break up complicated expressions that extend over multiple lines, but it will usually correctly indent such expressions which have already been broken up. Such an expression might end up looking like this:

```
x =
        (
            (Arbitrary parenthesized expression)
            +
            (
                (Parenthesized expression)
                *
                (Parenthesized expression)
            )
        );
```

Comments

Indent recognizes four kinds of comments. They are: straight text, "box" comments, UNIX-style comments, and comments that should be passed through unchanged. The action taken with these various types are as follows:

"Box" comments. *Indent* assumes that any comment with a dash immediately after the start of a comment (i.e. "/*–") is a comment surrounded by a box of stars. Each line of such a comment will be left unchanged, except that the first non-blank character of each successive line will be lined up with the beginning slash of the first line. Box comments will be indented (see below).

"Unix-style" comments. This is the type of section header which is used extensively in the UNIX system source. If the start of a comment ("/*") appears on a line by itself, indent assumes that it is a UNIX-style comment. These will be treated similarly to box comments, except the first non-blank character on each line will be lined up with the '*' of the "/*".

Unchanged comments. Any comment which starts in column 1 will be left completely unchanged. This is intended primarily for documentation header pages. The check for unchanged comments is made before the check for UNIX-style comments.

Straight text. All other comments are treated as straight text. *Indent* will fit as many words (separated by blanks, tabs, or newlines) on a line as possible. Straight text comments will be indented.

Comment indentation

Box comments, UNIX-style commnets, and straight text comments may be indented. If a comment is on a line with code, it will be started in the "comment column", which is set by the **–cnnn** command line parameter. Otherwise, if *nnn* is specified by the **–dnnn** command line parameter, the comment will be started at *nnn* indentation levels less than where code is currently being placed. (Indented comments will never be placed in column 1.) If the code on a line extends past the comment column, the comment will be moved to the next line.

DIAGNOSTICS

Diagnostic error messages are used to tell that a text line has been broken or is too long for the

output line.

FILES

.indent.pro profile file

BUGS

Does not know how to format "long" declarations.

NAME

install – install binaries

SYNTAX

/etc/install [– **c**] [– **m** mode] [– **o** owner] [– **g** group] [– **s**] binary destination

DESCRIPTION

Binary is moved (or copied if – **c** is specified) to *destination*. If *destination* already exists, it is removed before *binary* is moved. If the destination is a directory then *binary* is moved into the *destination* directory with its original file-name.

The mode for *Destination* is set to 755; the – **m** *mode* option may be used to specify a different mode.

Destination is changed to owner bin; the – **o** *owner* option may be used to specify a different owner.

Destination is changed to group bin; the – **g** *group* option may be used to specify a different group.

If the – **s** option is specified the binary is stripped after being installed.

Install refuses to move a file onto itself.

SEE ALSO

chgrp(1), chmod(1), cp(1), strip(1), chown(1)

NAME

invert – invert Ridge monochromatic display screen

SYNTAX

invert

DESCRIPTION

Invert reverses the contrast on the display screen. White characters on a black screen turn to black characters on a white screen, or vice versa. Screen memory bits are not modified; the display hardware handles the change. The current state of the screen remains fixed until inverted again. This interpretation remains in effect until the next *invert*(1).

NAME

iphostid – set or print identifier of current host system

SYNTAX

iphostid [identifier]

DESCRIPTION

The *iphostid* command prints the identifier of the current host in Internet dot notation. This numeric value is expected to be unique across all hosts and is normally set to the host's Internet address. The super-user can set the iphostid by giving an Internet address; this is usually done in the startup script **/etc/rc**. The Internet address can be found in the *hosts*(4) file.

NAME

join – relational database operator

SYNTAX

join [options] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is – , the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

Fields are normally separated by blank, tab or new-line. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

– **a***n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

– **e** *s* Replace empty output fields by string *s*.

– **j***n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.

– **o** *list* Each output line comprises the fields specifed in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number.

– **t***c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

awk(1), comm(1), sort(1).

BUGS

With default field separation, the collating sequence is that of **sort** – **b**; with – **t**, the sequence is that of a plain sort. The conventions of *join*, *sort*, *comm*, *uniq* and *awk*(1) are inconsistent.

NAME

kermit – file transfer, virt. terminal over tty link

SYNTAX

kermit c[**hilbe**] [*line*] [*baud*] [*esc*]

kermit r[**ddilb**] [*line*] [*baud*]

kermit s[**ddilb**] [*line*] [*baud*] *file* ...

DESCRIPTION

Kermit provides reliable file transfer and primitive virtual terminal communication between machines. It has been implemented on many different computers, including microprocessors (see below). The files transferred may be arbitrary ASCII data (7-bit characters) and may be of any length. The file transfer protocol uses small (96 character) checksummed packets, with ACK/NAK responses and timeouts. *Kermit* currently uses a ten second timeout and ten retries.

The arguments to *kermit* are a set of flags (no spaces between the flags), three optional args (which, if included, must be in the same order as the flags which indicate their presence), and, if this is a Send operation a list of one or more files. (It is similar in some way to the *tar* command structure).

Kermit has three modes, Connect, Send, and Receive. The first is for a virtual terminal connection, the other two for file transfer. These modes are specified by the first flag, which should be **c**, **s**, or **r**, respectively. Exactly one mode must be specified.

The **d** flag (debug) makes *kermit* a bit more verbose. The states *kermit* goes through are printed along with other traces of it's operation. A second **d** flag will cause *kermit* to give an even more detailed trace.

Use the **h** flag (half duplex) in connect mode to make *kermit* echo characters typed on the local keyboard, instead of assuming that the remote machine will do the echoing. This is necessary when communicating with half duplex systems like IBM CMS.

The **i** flag (image) allows slightly more efficient file transfer between UNIX machines. Normally (on Kermits defined to run on UNIX systems) newline is mapped to CRLF on output, CR's are discarded on input, and bytes are masked to 7 bits. If this is set, no mapping is done on newlines, and all eight bits of each byte are sent or received. Caution: this flag must not be set if the remote machine is one which sends parity (8 bit characters). This means IBM CMS systems.

The **l** flag (line) specifies the tty line that *kermit* should use to communicate with the other machine. This is specified as a regular filename, like "/dev/tty1". If no **l** option is specified, standard input is used and *kermit* assumes it is running on the remote host (ie. NOT the machine to which your terminal is attached).

The **b** flag (baud) sets the baud rate on the line specified by the **l** flag. No changes are made if the **b** flag is not used. Legal speeds are: 110, 150, 300, 1200, 2400, 4800, 9600, 19200.

The **e** flag (escape) allows the user to set the first character of the two character escape sequence for Connect mode. When the escape character is typed, *kermit* will hold it and wait for the next character. If the next character is **c** or **C**, *kermit* will close the connection with the remote host. If the second character is the same as the escape character, the escape character itself is passed. Any character other than these two results in a bell being sent to the user's terminal and no characters passed to the remote host. All other typed characters are passed through unchanged. The default escape character is '^'.

The file arguments are only meaningful to a Send *kermit*. The Receiving *kermit* will attempt to store the file with the same name that was used to send it. UNIX *kermits* normally convert outgoing file names to uppercase and incoming ones to lower case (see the **f** flag). If a filename

contains a slash (/) all outgoing kermits will strip off the leading part of the name through the last slash.

EXAMPLE

For this example we will assume two UNIX machines. We are logged onto "unixa" (the local machine), and want to communicate with "unixb" (the remote machine). There is a modem on "/dev/tty3".

We want to connect to "unixb", then transfer "file1" to that machine.

We type: kermit clb /dev/tty3 1200

Kermit answers: Kermit: connected...

Now we dial the remote machine and connect the modem. Anything typed on the terminal will be sent to the remote machine and any output from that machine will be displayed on our terminal. We press RETURN, get a "login:" prompt and login.

Now we need to start a *kermit* on the remote machine so that we can send the file over. First we start up the remote, (in this case receiving) *kermit*, then the local, (sending) one. Remember that we are talking to unixb right now.

We type: kermit r
(there is now a Receive *kermit* on unixb)

We type ^ (the kermit escape character (shift-6)) and then **c** to kill the local (Connecting) *kermit*.

Kermit answers: Kermit: disconnected.

We type: kermit slb /dev/tty3 1200 file1

Kermit answers: Sending file1 as FILE1

When the transmission is finished, *kermit* will type either "Send complete", or "Send failed.", depending on the success of the transfer. If we now wanted to transfer a file from unixb (remote) to unixa (local), we would use these commands:

```
kermit clb /dev/tty3 1200
  (connected to unixb)
kermit s file9
^c (up-arrow c not control-c)
  (talking to unixa again)
kermit rl /dev/tty3 1200
```

After all the transfers were done, we should connect again, log off of unixb, kill the Connect *kermit* and hang up the phone.

Detail on other implementations and on the protocol is given in the *Kermit Users Guide*, and the *Kermit Protocol Handbook*.

FEATURES

The KERMIT Protocol uses only printing ASCII characters, Ctrl-A, and CRLF. Ctrl-S/Ctrl-Q flow control can be used "underneath" the Kermit protocol (IXOFF line discipline on System V UNIX).

Since BREAK is not an ASCII character, *kermit* cannot send a BREAK to the remote machine. On some systems, a BREAK will be read as a NUL.

This *kermit* does have timeouts when run under UNIX, so the protocol is stable when communicating with "dumb" kermits (that don't have timeouts).

OTHER IMPLEMENTATIONS

See the *Kermit Users Guide.*

SEE ALSO

stty(1);
Kermit Users Guide, Fourth Edition (4 May 83), Frank da Cruz, Daphne Tzoar, Bill Catchings; *Kermit Protocol Manual,* Protocol Version 3 (29 April 83), Frank da Cruz, Bill Catchings. Both documents are from the Columbia University Center for Computing Activities, New York, New York, 10027, (212) 280-3703.

AUTHORS

KERMIT kernel by Bill Catchings, Columbia University Center for Computing Activities. KERMIT-UNIX adaptation by Chris Maio and Bob Cattani, Columbia University Computer Science Dept. Local mods for v6, System III, and System V by Walter Underwood. Includes bug fixes from Jim Guyton at RAND-UNIX. Enhanced for better use with System V by Ridge Computers.

DIAGNOSTICS

cannot open *device*
: The file named in the *line* argument did not exist or had the wrong permissions.

bad line speed
: The *baud* argument was not a legal speed.

Could not create *file*
: A Receive *kermit* could not create the file being sent to it.

nothing to connect to
: A Connect *kermit* was started without a *line* argument.

BUGS AND CAVEATS

There is no locking on the use of the outgoing line. Several users could run *kermit* (or anything else) on the line simultaneously.

The acronym (*K*L10 *E*rror-free *R*eciprocal *M*icro *I*nterconnect over *T*TY lines) is charming, but strained.

This implementation does not send or process error-message packets.

Eight-bit quoting is not implemented.

NAME

kill – terminate a process

SYNTAX

kill [– signo] PID ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with *&* is reported by the Shell. Process numbers can also be found by using *ps*(1).

The details of the kill are described in *kill*(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The killed process must belong to the current user unless he is the super-user.

If a signal number preceded by – is given as first argument, that signal is sent instead of terminate (see *signal*(2)). In particular "kill – 9 ..." is a sure kill.

SEE ALSO

ps(1), sh(1), kill(2), signal(2).

NAME

last – indicate last logins of users and teletypes

SYNTAX

last [– N] [name ...] [tty ...]

DESCRIPTION

Last will look back in the *wtmp* file which records all logins and logouts for information about a user, a teletype or any group of users and teletypes. Arguments specify names of users or teletypes of interest. Names of teletypes may be given fully or abbreviated. For example 'last 0' is the same as 'last tty0'. If multiple arguments are given, the information which applies to any of the arguments is printed. For example 'last root console' would list all of "root's" sessions as well as all sessions on the console terminal. *Last* will print the sessions of the specified users and teletypes, most recent first, indicating the times at which the session began, the duration of the session, and the teletype which the session took place on. If the session is still continuing or was cut short by a reboot, *last* so indicates.

The pseudo-user **reboot** logs in at reboots of the system, thus

last reboot

will give an indication of mean time between reboot.

Last with no arguments prints a record of all logins and logouts, in reverse order. The – N option limits the report to N lines.

If *last* is interrupted, it indicates how far the search has progressed in *wtmp*. If interrupted with a quit signal (generated by a control-\) *last* indicates how far the search has progressed so far, and the search continues.

FILES

/etc/wtmp login data base

SEE ALSO

utmp(4)

NAME

ld – link editor

SYNTAX

ld [option] ... file ...

DESCRIPTION

Ld combines several object programs into one, resolves external references, and searches libraries. In the simplest case, several object *files* are given, and *ld* combines them into an executable object module called **a.out**. If no errors occur during the load, the execute permission for **a.out** is set.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine (unless the – **e** option is specified).

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by *ranlib*(1), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. The first member of a library should be a file named '__.SYMDEF', which is understood to be a dictionary for the library as produced by *ranlib*(1); the dictionary is searched iteratively to satisfy as many references as possible.

The symbols '_etext', '_edata' and '_end' ('etext', 'edata' and 'end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. Do not define these symbols.

Ld understands several options. Except for – **l**, they should appear before the file names.

– **A** This option specifies incremental loading, i.e. linking is to be done in a manner so that the resulting object may be read into an already executing program. The next argument is the name of a file whose symbol table will be taken as a basis on which to define additional symbols. Only newly linked material will be entered into the text and data portions of **a.out**, but the new symbol table will reflect every symbol defined before and after the incremental load. This argument must appear before any other object file in the argument list. The – **T** option may be used as well, and will be taken to mean that the newly linked segment will commence at the corresponding address (which must be a multiple of 4096). The default value is the old value of _end.

– **C** Produce an executable file with shared code and data. The data segment is put at the first page boundary past the end of the code segment. The output file is given the magic number 9B01, which signals exec(2) (or spawn(2)) to cause the code and data segments to be mapped to the same address space.

– **D***dorg*
Set the data segment origin to *dorg*, which is hexadecimal. The default origin is 0.

– **d** Force definition of common storage even if the – **r** flag is present.

– **e** The following argument is taken to be the name of the entry point of the loaded program; location 0 is the default.

– **l***x* This option is an abbreviation for the library name '/lib/lib*x*.a', where *x* is a string. If that does not exist, *ld* tries '/usr/lib/lib*x*.a' A library is searched when its name is encountered, so the placement of a – **l** is significant.

– **M** produce a primitive load map, listing the names of the files which will be loaded.

– **m mapfile**
create a load map called *mapfile* (as described in map(1)).

- **– N** Do not make the text portion read only or sharable. (Use "magic number" 0407.)
- **– o** The *name* argument after **– o** is used as the name of the *ld* output file, instead of **a.out**.
- **– r** Generate relocation bits in the output file so that it can be the subject of another *ld* run. This flag also prevents final definitions from being given to common symbols, and suppresses the 'undefined symbol' diagnostics.
- **– S** 'Strip' the output by removing all symbols except locals and globals.
- **– s** Strip the output of the symbol table and relocation bits to save space (but impair the usefulness of the debuggers). This information can be removed with *strip*(1).
- **– T** The next argument is a hexadecimal number which sets the text segment origin. The default origin is the size of the **a.out** header (see **a.out**(4)).
- **– t** ("trace") Print the name of each file as it is processed.
- **– u** Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.
- **– V***hexarg* Set the a_version field of the header to *hexarg*, which is hexadecimal.
- **– W***hexarg* Set the a_type field of the header to *hexarg*, which is hexadecimal.
- **– X** Save local symbols except for those whose names begin with 'L'. This option is used by *cc*(1) to discard internally-generated labels while retaining symbols local to routines.
- **– x** Do not preserve local (non-global) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.
- **– y***sym* Indicate each file in which *sym* appears, its type and whether the file defines or references it. Many such options may be given to trace many symbols. (It is usually necessary to begin *sym* with an '_' because external C and FORTRAN variables begin with underscores.)

FILES

/lib/lib*.a	libraries
/usr/lib/lib*.a	more libraries
/usr/local/lib/lib*.a	still more libraries
a.out	output file

SEE ALSO

as(1), ar(1), cc(1), ranlib(1), map(1)

BUGS

There is no way to force data to be page-aligned.

NAME

leave – remind you when you have to leave

SYNTAX

leave [hhmm]

DESCRIPTION

Leave waits until the specified time, then reminds you that you have to leave. You are reminded 5 minutes and 1 minute before the actual time, at the time, and every minute thereafter. When you log off, *leave* exits just before it would have printed the next message.

The time of day is in the form hhmm where hh is a time in hours (on a 12 or 24 hour clock). All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

If no argument is given, *leave* prompts with "When do you have to leave?". A reply of newline causes *leave* to exit, otherwise the reply is assumed to be a time. This form is suitable for inclusion in a *.login* or *.profile*.

Leave ignores interrupts, quits, and terminates. To get rid of it you should either log off or use "kill – 9" giving its process id.

SEE ALSO

calendar(1)

BUGS

NAME

lex – generate programs for simple lexical tasks

SYNTAX

lex [– **rctvn**] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file **lex.yy.c** is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in **[abx– z]** to indicate **a**, **b**, **x**, **y**, and **z**; and the operators *****, **+**, and **?** mean respectively any non-negative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character **.** is the class of all ASCII characters except new-line. Parentheses for grouping and vertical bar for alternation are also supported. The notation $r\{d,e\}$ in a rule indicates between d and e instances of regular expression r. It has higher precedence than | but lower than *, ?, +, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character $ at the end of an expression requires a trailing new-line. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus **[a– zA– Z]+** matches a string of letters.

Three subroutines defined as macros are expected: **input()** to read a character; **unput(*c*)** to replace a character read; and **output(*c*)** to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named **yylex()**, and the library contains a **main()** which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function **yymore()** accumulates additional characters into the same *yytext*; and the function **yyless(*p*)** pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext*+ *yyleng*. The macros *input* and *output* use files **yyin** and **yyout** to read from and write to, defaulted to **stdin** and **stdout**, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the **lex.yy.c** file. All rules should follow a %% as in YACC. Lines preceding %% which begin with a non-blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

EXAMPLE

```
D       [0– 9]
%%
if      printf("IF statement\n");
[a– z]+ printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
                switch (input())
```

```
                    {
                    case '/': break;
                    case '*': unput('*');
                    default: go to loop;
                    }
            }
```

The external names generated by *lex* all begin with the prefix **yy** or **YY**.

The flags must appear before any files. The flag **– r** indicates RATFOR actions, **– c** indicates C actions and is the default, **– t** causes the **lex.yy.c** program to be written instead to standard output, **– v** provides a one-line summary of statistics of the machine generated, **– n** will not print out the – summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n* number of positions is *n* (default 2000)

%n *n* number of states is *n* (500)

%t *n* number of parse tree nodes is *n* (1000)

%a *n* number of transitions is *n* (3000)

The use of one or more of the above automatically implies the **– v** option, unless the **– n** option is used.

SEE ALSO

yacc(1).
Lex tutorial in the *ROS Programmer's Guide*

BUGS

The **– r** option is not yet fully operational.

NAME

line – read the first line of a file

SYNTAX

line

DESCRIPTION

Line copies the first line (everything up to the first new-line character) from the standard input and writes it on the standard output. It returns an exit code of 1 on EOF and always prints at least a new-line character. **Line** input/output can be re-directed. **Line** is often used in shell scripts to read from the user's terminal.

SEE ALSO

sh(1), read(2).

NAME

link – object module linker for pasc(1), fort(1), and rasm(1)

SYNTAX

link [-H] [-N] [-L listfile] [-O objectfile] file.O file ...

DESCRIPTION

Link combines several object files into one, resolves references between the modules, assigns virtual adresses to program and data references, and searches libraries to satisfy external references. The result is generally an executable file.

Input **files** are processed in the order specified. The first **file** must end with the **.o** extension (upper- or lowercase). The default name of the resulting executable file is the same as the **.o** file except without that extension.

Information internal to a file indicates whether it is a regular object file or a file of library routines. Only the routines actually referenced are linked to the result file. Other object files are linked in their entirety.

– h indicates that the output file will be suitable input for another subsequent linking. (Messages about external reference errors are suppressed and relocation information is saved until the subsequent use of the linker. (**H** stands for hex.) By default, the output is executable binary code.

– n specifies not to generate the 8-byte code file header at the beginning of the output objectfile. This header, of the form "9B 00 00 00 00 08 00 00", is a branch to location 8. ROS reads this header to distinguish the file between a binary executable file and a shell script executable file. In certain cases of binary system programs, the header should be left off.

– l listfile

specifies the name of a listfile to be generated. By default, no such file is generated.

– o filename

The **-o** option overrides the default and specifies the name of the resulting output file.

NAME

link, unlink – exercise link and unlink system calls

SYNTAX

/etc/link file1 file2
/etc/unlink file

DESCRIPTION

Link and *unlink* perform their respective system calls on their arguments, abandoning all error checking. These commands may only be executed by the super-user, who should know what he or she is doing.

SEE ALSO

rm(1), link(2), unlink(2).

NAME

lint – a C program checker

SYNTAX

lint [option] ... file ...

DESCRIPTION

Lint attempts to detect features of the C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with **.c** are taken to be C source files. Arguments whose names end with **.ln** are taken to be the result of an earlier invocation of *lint* with either the – **c** or the – **o** option used. The **.ln** files are analogous to **.o** (object) files that are produced by the *cc*(1) command when given a **.c** file as input. Files with other suffixes are warned about and ignored.

Lint will take all the **.c**,**.ln**, and **llib-l***x***.ln** (specified by – **l***x*) files and process them in their command line order. By default, *lint* appends the standard C lint library (**llib-lc.ln**) to the end of the list of files. However, if the – **p** option is used, the portable C lint library (**llib-port.ln**) is appended instead. When the – **c** option is not used, the second pass of *lint* checks this list of files for mutual compatibility. When the – **c** option is used, the **.ln** and the **llib-l***x***.ln** files are ignored.

Any number of *lint* options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

– **a** Suppress complaints about assignments of long values to variables that are not long.

– **b** Suppress complaints about **break** statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in many such complaints).

– **h** Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

– **u** Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program).

– **v** Suppress complaints about unused arguments in functions.

– **x** Do not report variables referred to by external declarations but never used.

The following arguments alter *lint*'s behavior:

– **l***x* Include additional lint library **llib-l***x***.ln**. For example, you can include a lint version of the Math Library **llib-lm.ln** by inserting – **lm** on the command line. This argument does not suppress the default use of **llib-lc.ln**. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

– **n** Do not check compatibility against either the standard or the portable lint library.

– **p** Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

– **c** Cause *lint* to produce a **.ln** file for every **.c** file on the command line. These **.ln** files are the product of *lint*'s first pass only, and are not checked for inter-function

compatibility.

– o lib Cause *lint* to create a lint library with the name **llib-l***lib***.ln**. The **– c** option nullifies any use of the **– o** option. The lint library produced is the input that is given to *lint*'s second pass. The **– o** option simply causes this file to be saved in the named lint library. To produce a **llib-l***lib***.ln** without extraneous messages, use of the **– x** option is suggested. The **– v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file **llib-lc** is written). These option settings are also available through the use of "lint comments" (see below).

The **– D**, **– U**, and **– I** options of *cpp*(1) and the **– g** and **– O** options of *cc*(1) are also recognized as separate arguments. The **– g** and **– O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the *cc*(1) command. Other options are warned about and ignored. The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for *lint*. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by *lint*.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
: at appropriate points stops comments about unreachable code. (This comment is typically placed just after calls to functions like *exit*(2)).

/*VARARGS*n*/
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
: turns on the **– v** option for the next function.

/*LINTLIBRARY*/
: at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **– v** and **– x** options.

Lint produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **– c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the **– c** and the **– o** options allows for incremental use of *lint* on a set of C source files. Generally, one invokes *lint* once for each source file with the **– c** option. Each of these invocations produces a **.ln** file which corresponds to the **.c** file, and prints all messages that are about just that source file. After all the source files have been separately run through *lint*, it is invoked once more (without the **– c** option), listing all the **.ln** files with the needed **– l***x* options. This will print all the inter-file inconsistencies. This scheme works well with *make*(1); it allows *make* to be used to *lint* only the source files that have been modified since the last time the set of source files were *lint*ed.

FILES

/usr/lib	the directory where the lint libraries specified by the **– l***x* option must exist
/usr/lib/lint[12]	first and second passes
/usr/lib/llib-lc.ln	declarations for C Library functions (binary format; source is in **/usr/lib/llib-lc**)
/usr/lib/llib-port.ln	declarations for portable functions (binary format; source is in **/usr/lib/llib-port**)
/usr/lib/llib-lm.ln	declarations for Math Library functions (binary format; source is in **/usr/lib/llib-lm**)

/usr/tmp/*lint* temporaries

SEE ALSO

cc(1), cpp(1), make(1).

BUGS

exit(2), *longjmp*(3C) *(partof setjmp(3C))*, and other functions that do not return are not understood; this causes various lies.

NAME

login – sign on

SYNTAX

login [name [env-var ...]]

DESCRIPTION

Login initiates a user session. It asks the user's id and usually his password.

Login can be invoked by a user to end the current session and start a new one under any other user name. When a user logs off by typing control-d, the system automatically invokes *login* in preparation for a new user.

If the user does not supply a user name as an argument, *login* asks for it. If the user has previously used the *passwd(1)* command to establish a password, *login* asks for the password. To prevent accidental release of the user's password, *login* turns off the character echo while it is being entered.

With dial-up access, *login* also prompts "dialup password:". Both passwords are required for a successful login.

If you do not complete the login successfully within a certain period of time (e.g., one minute), you are likely to be silently disconnected.

After a successful login, accounting files are updated, the procedure */etc/profile* is performed, the message-of-the-day file (**/etc/motd**) is printed, the user-ID, the group-ID, the working directory, and the command interpreter (usually *sh*(1)) is initialized, and the file **.profile** in the working directory is excuted, if it exists. These specifications are found in the **/etc/passwd** file entry for the user. The name of the command interpreter in use is "-", followed by the last component of the interpreter's pathname (i.e., – **sh**). If this field in the password file is empty, then the default command interpreter, */bin/sh* is used.

The basic *environment* (see *environ*(5)) is initialized to:

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
TZ=*timezone-specification*

The environment may be expanded or modified by supplying additional arguments to *login*, either at execution time or when *login* requests your login name. The arguments may take either the form *xxx* or *xxx=yyy*. Arguments without an equal sign are placed in the environment as

L*n*=xxx

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variables containing an = are placed into the environment without modification. If they already appear in the environment, then they replace the older value. There are two exceptions. The variables PATH and SHELL cannot be changed. This prevents people, logging into restricted shell environments, from spawning secondary shells which aren't restricted. Both *login* and *getty* understand simple single character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

FILES

/etc/utmp	accounting
/etc/wtmp	accounting
/usr/mail/*your-name*	mailbox for user *your-name*
/etc/motd	message-of-the-day
/etc/passwd	password file

/etc/profile	system profile
.profile	user's login profile

SEE ALSO

mail(1), newgrp(1), sh(1), su(1), passwd(4), profile(4), environ(5).

DIAGNOSTICS

No utmp entry. You must exec "login" from the lowest level "sh" if you attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

BUGS

Login may not be used as a command by this version of ROS. You must terminate the shell with an end of file (control-d) character.

NAME

logname – get login name

SYNTAX

logname

DESCRIPTION

Logname returns the contents of the environment variable $LOGNAME, which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1), logname(3X), environ(5).

NAME

look – find lines in a sorted list

SYNTAX

look [**–df**] string [file]

DESCRIPTION

Look consults a sorted *file* and prints all lines that begin with *string*. It uses binary search.

The options **d** and **f** affect comparisons as in *sort*(1):

d 'Dictionary' order: only letters, digits, tabs and blanks participate in comparisons.

f Fold. Upper case letters compare equal to lower case.

If no *file* is specified, */usr/dict/words* is assumed with collating sequence **–df.**

FILES

/usr/dict/words

SEE ALSO

sort(1), grep(1)

NAME

lorder – find ordering relation for an object library

SYNTAX

lorder file ...

DESCRIPTION

The input is one or more object or library archive (see *ar*(1)) *files*. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1).

This brash one-liner intends to build a new library from existing '.o' files.

```
ar cr library `lorder *.o |tsort`
```

The need for lorder may be vitiated by use of *ranlib*(1), which converts an ordered archive into a randomly accessed library.

FILES

*symref, *symdef

nm(1), sed(1), sort(1), join(1)

SEE ALSO

tsort(1), ld(1), ar(1), ranlib(1)

BUGS

The names of object files, in and out of libraries, must end with '.o'; nonsense results otherwise.

NAME

lpr, lprm, lpq, print – line printer spooler

SYNTAX

lpr [**–m**] [*name* ...] [**–b**] [**–c**] [**–r**]
lprm [*id* ...] [*filename* ...] [*owner* ...]
lpq
print [*file* ...]

DESCRIPTION

Lpr puts the named files into an output queue, and prints them when the output device is ready. **-m** causes notification of job completion to be sent by mail(1).

Lpr normally does not move or copy the files to the spool area, but they wait in the current directory for their turn to be printed. This means that a file should not be modified in the current directory if it has been spooled because the wrong version of the file may be printed.
-b prints the specified word as a banner. For example,

```
lpr -b test file.c
```

prints the word "test" as a large header before the file "file.c" is printed.
-c causes the named file to be copied to the spool area, using more disc space but allowing the original file to be modified independent of the spooled copy.
-r causes the named file to be moved to the spool area, and removed from the current directory altogether. This is a dangerous option; use it carefully.

Lpq displays information about all files in the output queue. For each file, **lpq** reports its owner, an id number, its size in characters, and its name.

Lprm removes a named file from the output queue. The file can be specified by its *id*, *filename*, or *owner*. Use a file's *id* from the **lpq** output to remove it from the queue with **lprm**.

Print prints a copy of each named file on the line printer. It is actually a one-line shell script: **pr $* ¦ lpr .**

FILES

/usr/spool/lpd/*	spool area
/usr/lib/lpd	line printer demons
/usr/lib/lpf	file to handle banners and underlining

SEE ALSO

pr(1)

NAME

ls – list contents of directories

SYNTAX

ls [**–acdfghilmqrstux1CFR**] [*name*] ...

l [options] [*name*] ...

lx [options] [*name*] ...

lf [options] [*name*] ...

lr [options] [*name*] ...

ll [options] [*name*] ...

DESCRIPTION

Ls lists the files that are members of the directory *name*, or lists the file name if *name* is a file. With no *name* argument, **ls** lists the files in the working directory. *Options* request more detailed information about the characteristics of the named directories or files.
lx is equivalent to **ls -x**
lf is equivalent to **ls -F**
lr is equivalent to **ls -R**
ll is equivalent to **ls -l**
If **ll** *is redirected to a file, such as* **ll > filelist**, *it is equivalent to* **ls -l**.
l is equivalent to **ls -m**

The output is normally sorted alphabetically by *name*.

Some *Options* control the format of the listed names. In the simplest case, filenames are listed in vertically-sorted columns.

a	e	i
b	f	j
c	g	k
d	h	l

The *options* are:

–l (lowercase "L") List each *name* in long format, with: access mode, number of links, owner, size in bytes, time of last modification, and name.

```
-rwxrw-rw- 1 bin2756 May 30 15:14 wowzer
-rw-rw-r-- 1 root304 Jun 14 15:42 yes.1
-rw-r---w- 1 root593 Jun 14 15:57 zero.1
```

–t Sort by time of last modification (latest first) instead of by name:

```
-rw-r---w- 1 root593 Jun 14 15:57 zero.1
-rw-rw-r-- 1 root304 Jun 14 15:42 yes.1
-rwxrw-rw- 1 bin2756 May 30 15:14 wowzer
```

–a Include all names, including the ones which begin with a "." or "..", which are normally

suppressed. **-a** is a default option for the super-user (**root**).

—s Give size in blocks for each entry. A block is 1024 bytes even though the minimum storage increment is 4096 bytes:

```
total 68
   4 docsched
  12 exmanlist
   4 getnumber
  16 goodfor
```

—d If argument is a directory, list only its name, not its contents (often used with **-l** to get the status of a directory).

—r Reverse the order of sort to get reverse alphabetic or oldest first, as appropriate.

—u Use time of last access, instead of last modification, for sorting (**—t**) and/or printing (**—l**).

—c Use time of file creation for sorting (**—t**) and/or printing (**—l**).

—i Print file identifier number in the first column of the report for each file listed.

—f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **—l**, **—t**, **—s**, and **—r**, and turns on **—a**; output is sorted in the same way as entries appear in the directory.

—g Give group ID instead of owner ID in a long listing.

—m Force stream output format.

—1 (digit "one") Force one entry per line output format, e.g., to a teletype.

—C Force multi-column output (e.g., to a file).

—q Force printing of non-graphic characters in file names as the character "?"; this normally happens only if the output device is a terminal.

—h Force printing of non-graphic characters to be in the \ddd notation, in octal.

—x Force columnar printing to be sorted across, rather than down ,the page; this is the default if the last character of the name by which the program is invoked is an "x".

—F Cause directories to be marked with a trailing "/", and executable files to be marked with a trailing '*'. This is the default if the last character of the name by which the program is invoked is an `f".

—R Recursively list subdirectories encountered.

The mode printed under the **—l** option consists of 11 characters that are interpreted as follows:

The first character is:

- **d** if the entry is a directory;
- **b** if the entry is a block-type special file;
- **c** if the entry is a character-type special file;
- **—** if the entry is an ordinary file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next refers to permissions of others in the user-group of the file; and the last refers to all others. Within each set, the three characters indicate permission to read, to write, or to execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
— if the indicated permission is *not* granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; similarly, the user-execute permission character is given as **s** if the file has set-user-ID mode. If the group- or user-execute character is **s**, the group- or user-execute character may or may not be set for execution. In other words, the **s** obscures the underlying execute character, which may be **x** or —.

When the sizes of the files in a directory are listed a total count of blocks is printed.

FILES

/etc/passwd to get user IDs for **ls -l**.
/etc/group to get group IDs for **ls -g**.

SEE ALSO

chmod(1).

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable since "ls -s" is considerably different from **ls -s | lpr**. On the other hand, not using this setting would almost certainly be detrimental to old shell scripts that use "ls".

Column widths are not good choices for terminals that can tab.

NAME

m4 – macro processor

SYNTAX

m4 [options] [files]

DESCRIPTION

M4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is – , the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

– **e** Operate interactively. Interrupts are ignored and the output is unbuffered. Using this mode requires a special state of mind.

– **s** Enable line sync output for the C preprocessor (#line ...)

– **B***int* Change the size of the push-back and argument collection buffers from the default of 4,096.

– **H***int* Change the size of the symbol table hash array from the default of 199. The size should be prime.

– **S***int* Change the size of the call stack from the default of 100 slots. Macros take three slots, and non-macro arguments take one.

– **T***int* Change the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any file names and before any – **D** or – **U** flags:

– **D***name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

– **U***name*
undefines *name*.

Macro calls have the form:

name(arg1,arg2, ..., argn)

The (must immediately follow the name of the macro. If the name of a defined macro is not followed by a (, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; $# is replaced by the number of

arguments; $* is replaced by a list of all the arguments separated by commas; $@ is like $*, but each argument is quoted (with the current quotes).

undefine removes the definition of the macro named in its argument.

defn returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

pushdef like *define*, but saves any previous definition.

popdef removes current definition of its argument(s), exposing the previous one if any.

ifdef if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word *unix* is predefined on the UNIX System versions of *m4*.

shift returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote change quote symbols to the first and second arguments. The symbols may be up to five characters long. *Changequote* without arguments restores the original values (i.e., ` ').

changecom change left and right comment markers from the default # and new-line. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes new-line. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert *m4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum returns the value of the current output stream.

dnl reads and discards characters up to and including the next new-line.

ifelse has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr returns the value of its argument decremented by 1.

eval evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, −, *, /, % ^ (exponentiation), bitwise &, |, ^, and ~; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len returns the number of characters in its argument.

index returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.

substr	returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.
translit	transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.
include	returns the contents of the file named in the argument.
sinclude	is identical to *include*, except that it says nothing if the file is inaccessible.
syscmd	executes the UNIX System command given in the first argument. No value is returned.
sysval	is the return code from the last call to *syscmd*.
maketemp	fills in a string of XXXXX in its argument with the current process ID.
m4exit	causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.
m4wrap	argument 1 will be pushed back at final EOF; example: m4wrap(`cleanup( ) ')
errprint	prints its argument on the diagnostic output file.
dumpdef	prints current names and definitions, for the named items, or for all if no arguments are given.
traceon	with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.
traceoff	turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

SEE ALSO

cc(1), cpp(1). and the M4 tutorial in the *ROS Utility Guide*

NAME

pdp11, u3b, vax, ridge – provide truth value about your processor type

SYNTAX

pdp11

u3b

vax

ridge

DESCRIPTION

The following commands will return a true value (exit code of 0) if you are on a processor that the command name indicates.

pdp11 True if you are on a PDP-11/45 or PDP-11/70.

u3b True if you are on a 3B20S.

vax True if you are on a VAX-11/750 or VAX-11/780.

ridge True if you are on a Ridge 32.

The commands that do not apply will return a false (non-zero) value. These commands are often used within *make*(1) makefiles and shell procedures to increase portability.

SEE ALSO

sh(1), test(1), true(1).

NAME

mail – interactive message processing system

SYNTAX

Mail [*options*] [*name...*]
mail [*options*] [*name...*]
mailx [*options*] [*name...*]

DESCRIPTION

The command *mail* provides a comfortable, flexible environment for sending and receiving messages electronically. When reading mail, *mail* provides commands to facilitate saving, deleting, and responding to messages. When sending mail, *mail* allows editing, reviewing and other modification of the message as it is entered. The mail command is identical to *mailx* as provided with AT&T System 5; the *mailx* command was derived from and is very similar to the 4.2 command *Mail*.

Incoming mail is stored in a standard file for each user, called the system *mailbox* for that user. When *mail* is called to read messages, the *mailbox* is the default place to find them. As messages are read, they are marked to be moved to a secondary file for storage, unless specific action is taken, so that the messages need not be seen again. This secondary file is called the *mbox* and is normally located in the user's HOME directory (see "MBOX" (ENVIRONMENT VARIABLES) for a description of this file). Messages remain in this file until deliberately removed.

On the command line, *options* start with a dash (–) and any other arguments are taken to be destinations (recipients). If no recipients are specified, *mail* will attempt to read messages from the *mailbox*. Command line options are:

–d	Turn on debugging output. Neither particularly interesting nor recommended.
–e	Test for presence of mail. *Mail* prints nothing and exits with a successful return code if there is mail to read.
–f [*filename*]	Read messages from *filename* instead of *mailbox*. If no *filename* is specified, the *mbox* is used.
–F	Record the message in a file named after the first recipient. Overrides the "record" variable, if set (see ENVIRONMENT VARIABLES).
–h *number*	The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.
–H	Print header summary only.
–i	Ignore interrupts. See also "ignore" (ENVIRONMENT VARIABLES).
–n	Do not initialize from the system default *mail.rc* file.
–N	Do not print initial header summary.
–r *address*	Pass *address* to network delivery software. All tilde commands are disabled.
–s *subject*	Set the Subject header field to *subject*.
–u *user*	Read *user*'s *mailbox*. This is only effective if *user*'s *mailbox* is not read protected.
–U	Convert *uucp* style addresses to internet standards. Overrides the "conv" environment variable.

When reading mail, *mail* is in *command mode*. A header summary of the first several messages is displayed, followed by a prompt indicating *mail* can accept regular commands (see COMMANDS below). When sending mail, *mail* is in *input mode*. If no subject is specified on the command line, a prompt for the subject is printed. As the message is typed, *mail* will read the message and store it in a temporary file. Commands may be entered by beginning a line with the

tilde (~) escape character, followed by a single command letter and optional arguments. See TILDE ESCAPES for a summary of these commands.

At any time, the behavior of *mail* is governed by a set of *environment variables*. These are flags and valued parameters which are set and cleared via the **set** and **unset** commands. See ENVIRONMENT VARIABLES below for a summary of these parameters.

Recipients listed on the command line may be of three types: login names, shell commands, or alias groups. Login names may be any network address, including mixed network addressing. If the recipient name begins with "escape pipe-symbol" (\|), the rest of the name is taken to be a shell command to pipe the message through. This provides an automatic interface with any program that reads the standard input, such as *lp*(1) for recording outgoing mail on paper. Alias groups are set by the **alias** command (see COMMANDS below) and are lists of recipients of any type.

Regular commands are of the form

[**command**] [*msglist*] [*arguments*]

If no command is specified in *command mode*, **print** is assumed. In *input mode*, commands are recognized by the escape character, and lines not treated as commands are taken as input for the message.

Each message is assigned a sequential number, and there is, at any time, the notion of a 'current' message, marked by a '>' in the header summary. Many commands take an optional list of messages (*msglist*) to operate on, which defaults to the current message. A *msglist* is a list of message specifications separated by spaces, which may include:

n	Message number **n**.
.	The current message.
^	The first undeleted message.
$	The last message.
*****	All messages.
n—m	An inclusive range of message numbers.
user	All messages from **user**.
/string	All messages with **string** in the subject line (case ignored).
:c	All messages of type *c*, where *c* is one of:

d	deleted messages
n	new messages
o	old messages
r	read messages
u	unread messages

Note that the context of the command determines whether this type of message specification makes sense.

Other arguments are usually arbitrary strings whose usage depends on the command involved. File names, where expected, are expanded via the normal shell conventions (see *sh*(1)). Special characters are recognized by certain commands and are documented with the commands below.

At start-up time, *mail* reads commands from a system-wide file (**/usr/lib/mail/mail.rc**) to initialize certain parameters, then from a private start-up file (**$HOME/.mailrc**) for personalized variables. Most regular commands are legal inside start-up files, the most common use being to set up initial display options and alias lists. The following commands are not legal in the start-up file: **!**, **Copy**, **edit**, **followup**, **Followup**, **hold**, **mail**, **preserve**, **reply**, **Reply**, **shell**, and **visual**. Any errors in the start-up file cause the remaining lines in the file to be ignored.

COMMANDS

The following is a complete list of *mail* commands:

!*shell-command*
Escape to the shell. See "SHELL" (ENVIRONMENT VARIABLES).

*comment*
Null command (comment). This may be useful in *.mailrc* files.

=
Print the current message number.

?
Prints a summary of commands.

alias *alias name ...*
group *alias name ...*
Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

alternates *name ...*
Declares a list of alternate names for your login. When responding to a message, these names are removed from the list of recipients for the response. With no arguments, **alternates** prints the current list of alternate names. See also "allnet" (ENVIRONMENT VARIABLES).

cd [*directory*]
chdir [*directory*]
Change directory. If *directory* is not specified, $HOME is used.

copy [*filename*]
copy [*msglist*] *filename*
Copy messages to the file without marking the messages as saved. Otherwise equivalent to the **save** command.

Copy [*msglist*]
Save the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the **Save** command.

delete [*msglist*]
Delete messages from the *mailbox*. If "autoprint" is set, the next message after the last one deleted is printed (see ENVIRONMENT VARIABLES).

discard [*header-field ...*]
ignore [*header-field ...*]
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." The fields are included when the message is saved. The **Print** and **Type** commands override this command.

dp [*msglist*]
dt [*msglist*]
Delete the specified messages from the *mailbox* and print the next message after the last one deleted. Roughly equivalent to a **delete** command followed by a **print** command.

echo *string ...*
Echo the given strings (like *echo*(1)).

edit [*msglist*]
Edit the given messages. The messages are placed in a temporary file and the "EDITOR" variable is used to get the name of the editor (see ENVIRONMENT VARIABLES). Default editor is *ed*(1).

exit
xit
Exit from *mail*, without changing the *mailbox*. No messages are saved in the *mbox* (see also **quit**).

file [*filename*]
folder [*filename*]
Quit from the current file of messages and read in the specified file. Several special characters are recognized when used as file names, with the following substitutions:
% the current *mailbox*.
%**user**
the *mailbox* for **user**.
the previous file.
& the current *mbox*.
Default file is the current *mailbox*.

folders
Print the names of the files in the directory set by the "folder" variable (see ENVIRONMENT VARIABLES).

followup [*message*]
Respond to a message, recording the response in a file whose name is derived from the author of the message. Overrides the "record" variable, if set. See also the **Followup**, **Save**, and **Copy** commands and "outfolder" (ENVIRONMENT VARIABLES).

Followup [*msglist*]
Respond to the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line is taken from the first message and the response is recorded in a file whose name is derived from the author of the first message. See also the **followup**, **Save**, and **Copy** commands and "outfolder" (ENVIRONMENT VARIABLES).

from [*msglist*]
Prints the header summary for the specified messages.

group *alias name ...*
alias *alias name ...*
Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

headers [*message*]
Prints the page of headers along with the specified message. The "screen" variable sets the number of headers per page (see ENVIRONMENT VARIABLES). See also the **z** command.

help
Prints a summary of commands.

hold [*msglist*]
preserve [*msglist*]
Holds the specified messages in the *mailbox*.

if *s|r*
mail-commands
else
mail-commands
endif
Conditional execution. An *s* following the *if* statement will execute the following *mail-commands*, up to an **else** or **endif**, if the program is in *send* mode. The *r* following the *if* statement causes the *mail-commands* to be executed only in *receive* mode. Useful in the *.mailrc* file.

ignore *header-field ...*
discard *header-field ...*
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." All fields are included when the message is saved. The **Print** and **Type** commands override this command.

list
Prints all commands available. No explanation is given.

mail *name ...*
Mail a message to the specified users.

mbox [*msglist*]
Arrange for the given messages to end up in the standard *mbox* save file when *mail* terminates normally. See "MBOX" (ENVIRONMENT VARIABLES) for a description of this file. See also the **exit** and **quit** commands.

next [*message*]
Go to next message matching *message*. A *msglist* may be specified, but in this case the first valid message in the list is the only one used. This is useful for jumping to the next message from a specific user, since the name would be taken as a command in the absence of a real command. See the discussion of *msglists* above for a description of possible message specifications.

pipe [*msglist*] [*shell-command*]
| [*msglist*] [*shell-command*]
Pipe the message through the given *shell-command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the "cmd" variable. If the "page" variable is set, a form feed character is inserted after each message (see ENVIRONMENT VARIABLES).

preserve [*msglist*]
hold [*msglist*]
Preserve the specified messages in the *mailbox*.

Print [*msglist*]
Type [*msglist*]
Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the **ig**nore command.

print [*msglist*]
type [*msglist*]
Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command specified by the "PAGER" variable. The default command is *pg*(1) (see ENVIRONMENT VARIABLES).

quit
Exit from *mail*, storing messages that were read in *mbox* and unread messages in the *mailbox*. Messages that have been explicitly saved in a file are deleted.

Reply [*msglist*]
Respond [*msglist*]
Send a response to the author of each message in the *msglist*. The subject line is taken from the first message. If "record" is set to a filename, the response is saved at the end of that file (see ENVIRONMENT VARIABLES).

reply [*message*]
respond [*message*]
Reply to the specified message, including all other recipients of the message. If "record" is set to a filename, the response is saved at the end of that file (see ENVIRONMENT VARIABLES).

Save [*msglist*]
Save the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the **C**opy, **f**ollowup, and **F**ollowup commands and "outfolder" (ENVIRONMENT VARIABLES).

save [*filename*]
save [*msglist*] *filename*
Save the specified messages in the given file. The file is created if it does not exist. The message is deleted from the *mailbox* when *mail* terminates unless "keepsave" is set (see also ENVIRONMENT VARIABLES and the **ex**it and **q**uit commands).

set
set *name*
set *name=string*
set *name=number*
Define a variable called *name*. The variable may be given a null, string, or numeric value. **Se**t by itself prints all defined variables and their values. See ENVIRONMENT VARIABLES for detailed descriptions of the *mail* variables.

shell
Invoke an interactive shell (see also "SHELL" (ENVIRONMENT VARIABLES)).

size [*msglist*]
Print the size in characters of the specified messages.

source *filename*
Read commands from the given file and return to command mode.

top [*msglist*]
Print the top few lines of the specified messages. If the "toplines" variable is set, it is taken as the number of lines to print (see ENVIRONMENT VARIABLES). The default is 5.

touch [*msglist*]
Touch the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the *mbox* upon normal termination. See **exit** and **quit**.

Type [*msglist*]
Print [*msglist*]
Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the **ignore** command.

type [*msglist*]
print [*msglist*]
Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command specified by the "PAGER" variable. The default command is *pg*(1) (see ENVIRONMENT VARIABLES).

undelete [*msglist*]
Restore the specified deleted messages. Will only restore messages deleted in the current mail session. If "autoprint" is set, the last message of those restored is printed (see ENVIRONMENT VARIABLES).

unset *name* ...
Causes the specified variables to be erased. If the variable was imported from the execution environment (i.e., a shell variable) then it cannot be erased.

version
Prints the current version and release date.

visual [*msglist*]
Edit the given messages with a screen editor. The messages are placed in a temporary file and the "VISUAL" variable is used to get the name of the editor (see ENVIRONMENT VARIABLES). The default editor is *vi*(1).

write [*msglist*] *filename*
Write the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the **save** command.

xit
exit

Exit from *mail*, without changing the *mailbox*. No messages are saved in the *mbox* (see also quit).

z[+|-]

Scroll the header display forward or backward one full screen. The number of headers displayed is set by the "screen" variable (see ENVIRONMENT VARIABLES).

TILDE ESCAPES

The following commands may be entered only from *input mode*, by beginning a line with the tilde escape character (~). See "escape" (ENVIRONMENT VARIABLES) for changing this special character.

~! *shell-command*

Escape to the shell.

~.

Simulate end of file (terminate message input).

~: *mail-command*
~_ *mail-command*

Perform the command-level request. Valid only when sending a message while reading mail.

~?

Print a summary of tilde escapes.

~A

Insert the autograph string "Sign" into the message (see ENVIRONMENT VARIABLES).

~a

Insert the autograph string "sign" into the message (see ENVIRONMENT VARIABLES).

~b *name ...*

Add the *name*s to the blind carbon copy (Bcc) list.

~c *name ...*

Add the *name*s to the carbon copy (Cc) list.

~d

Read in the *dead.letter* file. See "DEAD" (ENVIRONMENT VARIABLES) for a description of this file.

~e

Invoke the editor on the partial message. See also "EDITOR" (ENVIRONMENT VARIABLES).

~f [*msglist*]

Forward the specified messages. The messages are inserted into the message, without alteration.

~h

Prompt for Subject line and To, Cc, and Bcc lists. If the field is displayed with an initial value, it may be edited as if you had just typed it.

~i *string*

Insert the value of the named variable into the text of the message. For example, **~A** is equivalent to '**~i** Sign.'

~m [*msglist*]

Insert the specified messages into the letter, shifting the new text to the right one tab stop. Valid only when sending a message while reading mail.

~p

Print the message being entered.

~q

Quit from input mode by simulating an interrupt. If the body of the message is not null, the partial message is saved in *dead.letter*. See "DEAD" (ENVIRONMENT VARIABLES) for a description of this file.

~r *filename*
~< *filename*
~< !*shell-command*

Read in the specified file. If the argument begins with an exclamation point (!), the rest of the string is taken as an arbitrary shell command and is executed, with the standard output inserted into the message.

~s *string* ...

Set the subject line to *string*.

~t *name* ...

Add the given *name*s to the To list.

~v

Invoke a preferred screen editor on the partial message. See also "VISUAL" (ENVIRONMENT VARIABLES).

~w *filename*

Write the partial message onto the given file, without the header.

~x

Exit as with *~q*, except the message is not saved in *dead.letter*.

~| *shell-command*

Pipe the body of the message through the given *shell-command*. If the *shell-command* returns a successful exit status, the output of the command replaces the message.

ENVIRONMENT VARIABLES

The following are environment variables taken from the execution environment and are not alterable within *mail*.

HOME=*directory*
The user's base of operations.

MAILRC=*filename*
The name of the start-up file. Default is $HOME/.mailrc.

The following variables are internal *mail* variables. They may be imported from the execution environment or set via the **set** command at any time. The **unset** command may be used to erase variables.

allnet
All network names whose last component (login name) match are treated as identical. This causes the *msglist* message specifications to behave similarly. Default is **noallnet**. See also the **alternates** command and the "metoo" variable.

append
Upon termination, append messages to the end of the *mbox* file instead of prepending them. Default is **noappend.**

askcc
Prompt for the Cc list after message is entered. Default is **noaskcc**.

asksub
Prompt for subject if it is not specified on the command line with the **–s** option. Enabled by default.

autoprint
Enable automatic printing of messages after **delete** and **undelete** commands. Default is **noautoprint**.

bang
Enable the special-casing of exclamation points (!) in shell escape command lines as in *vi*(1). Default is **nobang**.

cmd=*shell-command*
Set the default command for the **pipe** command. No default value.

conv=*conversion*
Convert uucp addresses to the specified address style. The only valid conversion now is *internet*, which requires a mail delivery program conforming to the RFC822 standard for electronic mail addressing. Conversion is disabled by default. See also "sendmail" and the **–U** command line option.

crt=*number*
Pipe messages having more than *number* lines through the command specified by the value of the "PAGER" variable (*pg*(1) by default). Disabled by default.

DEAD=*filename*
The name of the file in which partial letters are saved in the event of untimely interrupt or delivery errors. Default is $HOME/dead.letter.

debug
Enable verbose diagnostics for debugging. Messages are not delivered. Default is **nodebug**.

dot
Take a period on a line by itself during input from a terminal as end-of-file. Default is **nodot**.

EDITOR=*shell-command*
The command to run when the **edit** or **~e** command is used. Default is *ed*(**1**).

escape=*c*
Substitute *c* for the ~ escape character.

folder=*directory*
The directory for saving standard mail files. User specified file names beginning with a plus (+) are expanded by preceding the filename with this directory name to obtain the real filename. If *directory* does not start with a slash (/), $HOME is prepended to it. In order to use the plus (+) construct on a *mail* command line, "folder" must be an exported *sh* environment variable. There is no default for the "folder" variable. See also "outfolder" below.

header
Enable printing of the header summary when entering *mail*. Enabled by default.

hold
Preserve all messages that are read in the *mailbox* instead of putting them in the standard *mbox* save file. Default is **nohold**.

ignore
Ignore interrupts while entering messages. Handy for noisy dial-up lines. Default is **noignore**.

ignoreeof
Ignore end-of-file during message input. Input must be terminated by a period (.) on a line by itself or by the **~.** command. Default is **noignoreeof**. See also "dot" above.

keep
When the *mailbox* is empty, truncate it to zero length instead of removing it. Disabled by default.

keepsave
Keep messages that have been saved in other files in the *mailbox* instead of deleting them. Default is **nokeepsave**.

MBOX=*filename*
The name of the file to save messages which have been read. The **xit** command overrides this function, as does saving the message explicitly in another file. Default is $HOME/mbox.

metoo
If your login appears as a recipient, do not delete it from the list. Default is **nometoo**.

LISTER=*shell-command*
The command (and options) to use when listing the contents of the "folder" directory. The default is *ls*(1).

onehop
When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away).

outfolder
Causes the files used to record outgoing messages to be located in the directory specified by the "folder" variable unless the pathname is absolute. Default is **nooutfolder**. See "folder" above and the **S**ave, **C**opy, **f**ollowup, and **F**ollowup commands.

page
Used with the **p**ipe command to insert a form feed after each message sent through the pipe. Default is **nopage**.

PAGER=*shell-command*
The command to use as a filter for paginating output. This can also specify the options to be used. Default is *pg*(1).

prompt=*string*
Set the *command mode* prompt to *string*. Default is "? ".

quiet
Refrain from printing the opening message and version when entering *mail*. Default is **noquiet**.

record=*filename*
Record all outgoing mail in *filename*. Disabled by default. See also "outfolder" above.

save
Enable saving of messages in *dead.letter* on interrupt or delivery error. See "DEAD" for a description of this file. Enabled by default.

screen=*number*
Sets the number of lines in a screen–full of headers for the **h**eaders command.

sendmail=*shell-command*
Alternate command for delivering messages. Default is *mail*(1).

sendwait
Wait for background mailer to finish before returning. Default is **nosendwait**.

SHELL=*shell-command*
The name of a preferred command interpreter. Default is *sh*(1).

showto
When displaying the header summary and the message is from you, print the recipient's

name instead of the author's name.

sign=*string*
The variable inserted into the text of a message when the **~a** (autograph) command is given. No default (see also **~i** (TILDE ESCAPES)).

Sign=*string*
The variable inserted into the text of a message when the **~A** command is given. No default (see also **~i** (TILDE ESCAPES)).

toplines=*number*
The number of lines of header to print with the **top** command. Default is 5.

VISUAL=*shell-command*
The name of a preferred screen editor. Default is *vi*(1).

FILES

/bin/mail	
/bin/mailx	
/bin/Mail $HOME/.mailrc	personal start-up file
$HOME/mbox	secondary storage file
/usr/mail/*	post office directory
/usr/lib/mail/mail.help*	help message files
/usr/lib/mail/mail.rc	global start-up file
/tmp/R[emqsx]*	temporary files

SEE ALSO
binmail(1), pg(1), ls(1).

BUGS
Where *shell-command* is shown as valid, arguments are not always allowed. Experimentation is recommended.

Internal variables imported from the execution environment cannot be **unset**.

The full internet addressing is not fully supported by *mail*. The new standards need some time to settle down.

Attempts to send a message having a line consisting only of a "." are treated as the end of the message by *binmail*(1) (the local mail delivery program).

Error messages name the program mailx.

Help files and *rmmail* are contained in the */usr/lib/mailx* directory.

NAME

mailx – interactive message processing system

SYNTAX

Mail [*options*] [*name...*]
mail [*options*] [*name...*]
mailx [*options*] [*name...*]

DESCRIPTION

See *mail*(1) for description.

NAME

make – maintain, update, and regenerate groups of programs

SYNTAX

make [– **f** makefile] [– **p**] [– **i**] [– **k**] [– **s**] [– **r**] [– **n**] [– **b**] [– **e**] [– **m**] [– **t**] [– **d**] [– **q**] [names]

DESCRIPTION

The following is a brief description of all options and some special names:

– **f** *makefile* Description file name. *Makefile* is assumed to be the name of a description file. A file name of – denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

– **p** Print out the complete set of macro definitions and target descriptions.

– **i** Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

– **k** Abandon work on the current entry, but continue on other branches that do not depend on that entry.

– **s** Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

– **r** Do not use the built-in rules.

– **n** No execute mode. Print commands, but do not execute them. Even lines beginning with an @ are printed.

– **b** Compatibility mode for old makefiles.

– **e** Environment variables override assignments within makefiles.

– **t** Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

– **d** Debug mode. Print out detailed information on files and times examined.

– **q** Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the – **s** option.

.IGNORE Same effect as the – **i** option.

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no – **f** option is present, **makefile**, **Makefile**, **s.makefile**, and **s.Makefile** are tried in order. If *makefile* is – , the standard input is taken. More than one – *makefile* argument pair may appear.

Make updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out of date.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a :, then a (possibly null) list of prerequisite files or dependencies. Text following a ; and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><new-line> sequence. Everything printed by make (except the initial

tab) is passed directly to the shell as is. Thus,

```
echo a\
b
```

will produce

```
ab
```

exactly the same as the shell would.

Sharp (#) and new-line surround comments.

The following *makefile* says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**) and a common file **incl.h**:

```
pgm:    a.o b.o
        cc a.o b.o - o pgm
a.o:    incl.h a.c
        cc - c a.c
b.o:    incl.h b.c
        cc - c b.c
```

Command lines are executed one at a time, each by its own shell. The first one or two characters in a command can be the following: - , @, - @, or @- . If @ is present, printing of the command is suppressed. If - is present, *make* ignores an error. A line is printed when it is executed unless the - **s** option is present, or the entry **.SILENT:** is in *makefile*, or unless the initial character sequence contains a @. The - **n** option specifies printing without execution; however, if the command line has the string $(MAKE) in it, the line is always executed (see discussion of the MAKEFLAGS macro under *Environment*). The - **t** (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate *make*. If the - **i** option is present, or the entry **.IGNORE:** appears in *makefile*, or the initial character sequence of the command contains - . the error is ignored. If the - **k** option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The - **b** option allows old makefiles (those written for the old version of *make*) to run without errors. The difference between the old version of *make* and this version is that this version requires all dependency lines to have a (possibly null or implicit) command associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

Interrupt and quit cause the target to be deleted unless the target is a dependency of the special name .PRECIOUS.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The - **e** option causes the environment to override the macro assignments in a makefile.

The MAKEFLAGS environment variable is processed by *make* as containing any legal input option (except - **f**, - **p**, and - **r**) defined for the command line. Further, upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, MAKEFLAGS always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the - **n** option is used, the command $(MAKE) is executed anyway; hence, one can perform a **make** - **n** recursively on a whole software system to see what would have been executed. This is because the - **n** is put in MAKEFLAGS and passed to further invocations of $(MAKE). This is one way of debugging all of the makefiles for a software project without actually doing

anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. *String2* is defined as all characters up to a comment character or an unescaped newline. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional :*subst1*=*subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of *subst1* in the named macro are replaced by *subst2*. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, new-line characters, and beginnings of lines. An example of the use of the substitute sequence is shown under *Libraries*.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

$* The macro $* stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@ The $@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$< The $< macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out of date with respect to the target (i.e., the "manufactured" dependent file name). Thus, in the **.c.o** rule, the $< macro would evaluate to the **.c** file. An example for making optimized **.o** files from **.c** files is:

```
.c.o:
        cc - c - O $*.c
```

or:

```
.c.o:
        cc - c - O $<
```

$? The $? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules which must be rebuilt.

$% The $% macro is only evaluated when the target is an archive library member of the form **lib(file.o)**. In this case, $@ evaluates to **lib** and $% evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, $(@D) refers to the directory part of the string $@. If there is no directory part, ./ is generated. The only macro excluded from this alternative form is $?. The reasons for this are debatable.

Suffixes

Certain names (for instance, those ending with **.o**) have inferable prerequisites such as **.c**, **.s**, etc. If no update commands for such a file appear in *makefile*, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
.f .f~ .f.o .f~.o .f~.f
```

To print out the rules compiled into the *make* on any machine, use the following:

```
make - fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the **(null)** string which *printf*(3S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file (see *sccsfile*(4)). Thus, the rule **.c~.o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix it is incompatible with *make*'s suffix point-of-view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e. **.c:**) is the definition of how to build *x* from *x***.c**. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite. The default list is:

```
.SUFFIXES: .o .c .y .l .s
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o - o pgm
a.o b.o: incl.h
```

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*(1), *lex*(1), and *yacc*(1) respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix **.o** from a file with suffix **.c** is specified as an entry with **.c.o:** as the target and no dependents. Shell commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parenthesis, it is assumed to be an archive library, the string within parenthesis referring to a member within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library which contains **file.o**. (This assumes the LIB macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form **.*XX*.a** where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(file.o)** depend upon **file.o** explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date
.c.a:
        $(CC) - c $(CFLAGS) $<
```

```
        ar rv $@ $*.o
        rm - f $*.o
```

In fact, the **.c.a** rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) - c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?   @echo lib is now up to date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object file names (inside **lib**) whose C source files are out of date. The substitution mode translates the **.o** to **.c**. (Unfortunately, one cannot as yet transform to **.c**~; however, this may become possible in the future.)Note also, the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

[Mm]akefile and s.[Mm]akefile

SEE ALSO

sh(1).
Make- A Program for Maintaining Computer Programs and *An Augmented Version of Make* in *A Programmers Guide to ROS* .

BUGS

Some commands return non-zero status inappropriately; use **- i** to overcome the difficulty. Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*. The syntax **lib(file1.o file2.o file3.o)** is illegal. You cannot build **lib(file.o)** from **file.o**. The macro **$(a:.o=.c~)** doesn't work.

NAME

makekey – generate encryption key

SYNTAX

/usr/lib/makekey

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e., to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, ., /, and upper- and lower-case letters. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4,096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but broken in 4,096 different ways. Using the *input key* as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 *output key* bits in the result.

Makekey is intended for programs that perform encryption (e.g., *ed*(1) and *crypt*(1)).

SEE ALSO

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

NAME

man – find manual information by keywords; print the manual

SYNTAX

man –k *keyword ...*
man –f *file ...*
man [–] [*section*] *title ...*

DESCRIPTION

Man prints one-line descriptions of ROS commands, searches for manual pages according to a given keyword, and prints pages of this ROS Reference Manual.

–k *keywords*
prints a one-line description of each manual entry whose table of contents listing contains the keyword.

–f *filenames*
lists the table of contents lines for manual entries relating to those files.

Without **–k** or **–f, man** formats a specified set of manual pages. If a section specifier is given, **man** looks in that section of the manual for the given *titles. Section* is a section number (**1** thru **7**). The number may be followed by a single letter classifier (3M for instance) indicating a math library from section 3. If *section* is omitted, **man** searches all sections of the manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any.

If the standard output is a teletype, or if the flag – is given, **man** pipes its output through *cat*(1) with the **–s** option to delete useless blank lines, and through *more*(1) to stop after each page on the screen. Press the SPACE bar to continue, or control-D to scroll more lines when the output stops.

FILES

/usr/man/cat?/*

SEE ALSO

more(1), ul(1), whereis(1)

BUGS

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter, but some information is lost on a typewriter.

NAME

map – produce loadmap of an executable file

SYNTAX

map *file*

DESCRIPTION

Map creates the same loadmap of the executable *file* as the **-m** option of **ld(1)**.

By default, map produces a list of global (external) symbols only. **– l** (lowercase "L") causes a mapping of all symbols.

SEE ALSO

ld(1)

NAME

mesg – permit or deny messages

SYNTAX

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

DESCRIPTION

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

FILES

/dev/tty*

SEE ALSO

write(1).

DIAGNOSTICS

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

NAME

mkdir – make a directory

SYNTAX

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

mknod – build special file

SYNTAX

/etc/mknod name **c** | **b** major minor

DESCRIPTION

Mknod makes a directory entry for a special file. The first argument is the *name* of the entry. The second argument is **b** (which stands for "block") if the special file has an internal directory structure (like floppy discs), or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number), which may be either decimal or octal.

See the **DEVICE CODE** paragraph of each entry in section (7) for information on each device documented there.

EXAMPLE

According to the fl(7) page:

device type	*major*	*minor*	*block-type or character-type*	*standard file name*
floppy disc	3	0	block	/dev/fl

The floppy disc entry was created by:

```
mknod /dev/fl b 3 0
```

SEE ALSO

mknod(2).

NAME

mkstr – create an error message file by massaging C source

SYNTAX

mkstr [–] messagefile prefix file ...

DESCRIPTION

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly paged in and out.

Mkstr will process each of the specified *files,* placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. A typical usage of *mkstr* would be

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx.*

To process the error messages in the source to the message file, *mkstr* searches for the string 'error("' in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a new-line character; the null character terminates the message so it can be easily used when retrieved. The new-line character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file, which can be used to retrieve the message, i.e.:

```
char    efilname[] =  "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
{
        char buf[256];

        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
oops:
                        perror(efilname);
                        exit(1);
                }
        }
        if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

The optional – causes the error messages to be placed at the end of the specified message file for recompiling part of a large *mkstr*ed program.

SEE ALSO

lseek(2), xstr(1)

NAME

more, page – peruse a file on the screen

SYNTAX

more [– **cdflsu**] [– *n*] [+*linenumber*] [+/*pattern*] [name ...]

page *more options*

DESCRIPTION

More is a filter which allows examination of a continuous text one screenful at a time on a soft-copy terminal. It normally pauses after each screenful, printing --More-- at the bottom of the screen. If the user then types a carriage return, one more line is displayed. If the user presses the keyboard space-bar, another screenful 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 screenful. This is useful if *more* is being used as a filter in some setting, such as a class, where many users may be unsophisticated.

– **f** This causes *more* to count logical, rather than screen lines. That is, long lines are not folded. This option is recommended if *nroff* output is being piped through *ul*, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen 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 screenful 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 screenful is printed (but only if a full screenful is being printed), and *k* – 1 rather than *k* – 2 lines are printed in each screenful, where *k* is the number of lines the terminal can display.

More looks in the file */etc/termcap* to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22

lines.

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 screenful if no argument is given)

^D : display 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i*.

d : same as ^D (control-D)

*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 screenful of lines

*i*f : skip *i* screenfuls and print a screenful of lines

q or Q : Exit from *more*.

= : Display the current line number.

v : Start up the editor *vi* at the current line.

h : Help command; give a description of all the *more* commands.

i/expr : search for the *i*-th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

*i*n : search for the *i*-th occurrence of the last regular expression entered.

' : (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!command
: invoke a shell with *command*. The characters '%' and '!' in "command" are replaced with the current file name and the previous shell command respectively. If there is no current file name, '%' is not expanded. The sequences "\%" and "\!" are replaced by "%" and "!" respectively.

i:n : skip to the *i*-th next file given in the command line (skips to last file if n doesn't make sense)

i:p : skip to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

:f : display the current file name and line number.

:q or :Q

exit from *more* (same as q or Q).

(dot) repeat the previous command.

The commands take effect immediately, i.e., it is not necessary to press RETURN. Up to the time when the command character itself is given, the user may press the line-kill character to cancel the numerical argument being formed. In addition, use the erase character to redisplay the --More--(xx%) message.

At any time when output is being sent to the terminal, the user can press 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

csh(1), man(1), sh(1), environ(5)

NAME

mount, umount – mount and dismount file system

SYNTAX

/etc/mount [special directory [– **r**]]

/etc/umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The *directory* must exist already; it becomes the name of the root of the newly mounted file system.

These commands maintain a table of mounted devices. If invoked with no arguments, *mount* prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed. UMOUNT is NOT CURRENTLY AVAILABLE.

To mount a file system automatically at system boot time, insert a line in **/etc/inittab** of the following form:

MNT0::respawn:/etc/mount *disc-name directory-name*

FILES

/etc/mnttab mount table

SEE ALSO

setmnt(1), mount(2), mnttab(4).

DIAGNOSTICS

Mount issues a warning if the file system to be mounted is currently mounted under another name.

Umount complains if the special file is not mounted or if it is busy. The file system is busy if it contains an open file or some user's working directory.

BUGS

Curently, a tape cannot be mounted, and physical write-protect cannot be set. Some degree of validation is done on the file system, however it is generally unwise to mount garbage file systems.

NAME

msgs — system messages and junk mail program

SYNTAX

msgs [**–fhlopq**] to read messages
msgs -s to place messages
msgs -c [-days] to clean up old]

DESCRIPTION

Msgs is used to read system messages. These messages are sent by using the '-s' option and should be short pieces of information which are suitable to be read once by most users of the system.

Msgs will be invoked each time you login when it is placed in the file *.login (.profile* if you use */bin/sh*). It will then prompt you with the source and subject of each new message. If there is no subject line, the first few non-blank lines of the message will be displayed. If there is more to the message, you will be told how long it is and asked whether you wish to see the rest of the message. The possible responses are:

y type the rest of the message

RETURN
synonym for **y**.

n skip this message and go on to the next message.

– redisplay the last message.

p pipe message through *more*(1) and print.

P turn off piping through *more*(1) and print message.

q drops you out of *msgs;* the next time you run the program it will pick up where you left off.

x same as **q**.

s append the current message to the file "Messages" in the current directory; 's–' will save the previously displayed message. A 's' or 's–' may be followed by a space and a filename to receive the message replacing the default "Messages".

m or 'm–' causes a copy of the specified message to be placed in a temporary mailbox and *mail*(1) to be invoked on that mailbox. Both 'm' and 's' accept a numeric argument in place of the '–'.

Msgs keeps track of the next message you will see by a number in the file *.msgsrc* in your home directory. If the *.msgrc* file does not exist, *msgs* will create it. In the directory */usr/msgs*, it keeps a set of files whose names are the (sequential) numbers of the messages they represent. The file */usr/msgs/bounds* shows the low and high number of the messages in the directory so that *msgs* can quickly determine if there are no messages for you. If the contents of *bounds* is incorrect, it can be fixed by removing it; *msgs* will make a new *bounds* file the next time it is run.

Options to msgs include:

-s Places messages in */usr/msgs* directory, in preparation for reading by other users. The **-s** option will prompt you for the subject and text of your message. A <*ctrl*> *d* allows you to exit.

-c [-days]
Can only be invoked by root or the daemon. Defaults to removing all files 21 days or older. Any number of days can be specified.

-o Before exiting, *msgs* will prompt you for any additional commands.

–f causes it not to say "No new messages.". This is useful in your *.login* file since this is often the case here.

–q Queries whether there are messages, printing "There are new messages." if there are. The command "msgs –q" is often used in login scripts.

–h causes *msgs* to print the first part of messages only.

–l option causes only locally originated messages to be reported.

num A message number can be given on the command line, causing *msgs* to start at the specified message, rather than at the next message indicated by your *.msgsrc* file. Thus

msgs –h 1

prints the first part of all messages.

–number

will cause *msgs* to start the *number* of messages back from the one indicated by your *.msgsrc* file. This is useful for reviews of recent messages.

–p causes long messages to be piped through *more*(1).

Within *msgs*, you can also go to any specific message by typing its number when *msgs* requests input as to what to do.

FILES

/usr/msgs/*	database
~/.msgsrc	number of next message to be presented

SEE ALSO

mail(1), more(1), news(1)

NAME

mt – magnetic tape manipulating program

SYNTAX

mt [– **f** *tapename*] *command* [*count*]

DESCRIPTION

Mt passes certain commands to a magnetic tape drive. If *tapename* is not specified, **mt** sends the command to the device named by the environment variable TAPE. If TAPE does not exist, commands go to the device **/dev/rmt0**.

Tapename must be a raw (non-block) tape device. By default, **mt** performs the command once. *count* may be entered to perform the *command* multiple times.

Only as many characters as are required to uniquely identify a *command* need be specified. The possible commands are:

eof, weof
Write *count* end-of-file marks at the current position on the tape.

fsf Forward space *count* files.

fsr Forward space *count* records.

bsf Back space *count* files.

bsr Back space *count* records.

rewind Rewind the tape (*Count* is ignored.)

offline, rewoffl
Rewind the tape and place the tape unit off-line (*Count* is ignored.)

status Print status information about the tape unit.

hispeed
set tape drive operation to high speed for this invocation of *mt* only.

lowspeed
set tape drive operation to low speed for this invocation of *mt* only.

Mt returns a 0 exit status when the operation(s) were successful, 1 if the command was unrecognized, and 2 if an operation failed.

FILES

/dev/rmt* Raw magnetic tape interface

SEE ALSO

dd(1), ioctl(2), mt(7)

NAME

mvdir – move a directory

SYNTAX

/etc/mvdir dirname name

DESCRIPTION

Mvdir renames directories within a file system. *Dirname* must be a directory; *name* must not exist. Neither name may be a sub-set of the other (**/x/y** cannot be moved to **/x/y/z**, nor vice versa).

Only super-user can use *mvdir*.

SEE ALSO

NAME

newaliases – rebuild the data base for the mail aliases file

SYNTAX

newaliases

DESCRIPTION

Newaliases rebuilds the random access data base for the mail aliases file */usr/lib/aliases*. It must be run each time */usr/lib/aliases* is changed in order for the change to take effect.

NOTE

/usr/lib/aliases.dir and */usr/lib/aliases.pag* must exist. Enter:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
```

SEE ALSO

aliases(4), sendmail(1)

NAME

newform – change the format of a text file

SYNTAX

newform [– **s**] [– **i**tabspec] [– **o**tabspec] [– **b**n] [– **e**n] [– **p**n] [– **a**n] [– **f**] [– **c**char] [– **l**n] [files]

DESCRIPTION

Newform reads lines from the named *files*, or the standard input if no input file is named, and reproduces the lines on the standard output. Lines are reformatted in accordance with command line options in effect.

Except for – **s**, command line options may appear in any order, may be repeated, and may be intermingled with the optional *files*. Command line options are processed in the order specified. This means that option sequences like "– **e**15 – **l**60" will yield results different from "– **l**60 – **e**15". Options are applied to all *files* on the command line.

– **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

SYNTAX

newgrp [–] [*group-ID-num*]

DESCRIPTION

Newgrp changes the group identification of the user who invokes it. After executing newgrp, the same user remains logged in, and the working directory remains the same, but the user becomes a member of *group-ID-num* and has group access to the files belonging to members of that group.

The members of *group-ID-num* are those who share that group number in **/etc/passwd**, or who are listed as members of that group-id-num in **/etc/group**.

Newgrp without an argument changes the group identification to the user's initial log-in group as recorded 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.

A password is demanded if the group has a password and the user himself does not, or if the group has a password and the user is not listed in **/etc/group** as being a member of that group.

When most users log in, they are members of the group named **other**.

FILES

/etc/group
/etc/passwd

SEE ALSO

login(1), group(4).

BUGS

There is no convenient way to enter a password into **/etc/group**. Use of group passwords is not encouraged, because, by their very nature, they encourage poor security practices. Group passwords may disappear in the future.

NAME

news – print news items

SYNTAX

news [– **a**] [– **n**] [– **s**] [items]

DESCRIPTION

News informs the users about current events, as recorded in the **/usr/news** directory.

If executed without arguments, **news** prints all files in **/usr/news**, the newest one first, with appropriate headers. **News** stores the current time in the user's home directory (in a file named **.news_time**) so that it will later know which news items are current for the user and which items the user has already seen.

– **a** option causes *news* to print all items, regardless of currency. In this case, the stored time is not changed.

– **n** causes *news* to report the names of the current items without printing their contents, and without changing the stored time.

– **s** causes *news* to report how many current items exist, without printing their names or contents, and without changing the stored time. It is useful to include **news** – **w** in a user's **.profile** file, or in the system **/etc/profile**, so that each user is notified at log-in time if he has new news.

Any other arguments are taken to be the names of specific news itmes that are to be printed.

To stop the printing of a news item, press DELETE, and the next item will start. To stop all printing, press DELETE twice within one second and **news** will terminate.

FILES

/etc/profile
/usr/news/*
$HOME/.news_time

SEE ALSO

profile(4), environ(5).

NAME

nice – run a command at lower priority

SYNTAX

nice [– increment] command [arguments]

DESCRIPTION

A high nice number results in low priority. Low numbers result in high priority. **Nice** executes *command* with a lower scheduling priority by adding *increment* to its priority value.

The normal priority of a user process is 20. If *increment* is omitted, a value of 10 is assumed. thereby setting the actual priority to 20+ 10, a lower priority than 20.

Priorities 0 to 8 are reserved for system processes, 9 to 19 are for device drivers, and 20 to 99 are for users. An *increment* greater than 79 is taken to be equal to 79.

Only the super-user may specify a negative *increment*, which would reduce the priority value to yield a higher priority. e.g., – – **10**.

SEE ALSO

nohup(1), nice(2).

DIAGNOSTICS

Nice returns the exit status of the *command*.

NAME

nl – line numbering filter

SYNTAX

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 supressed; **rn**, right justified, leading zeroes supressed; **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

SYNTAX

nm [– **agnopru**] [file ...]

DESCRIPTION

Nm prints the name list (symbol table) of each object *file* in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no *file* is given, the symbols in "a.out" are listed.

Each symbol name is preceded by its value (blanks if undefined) and one of the letters U (undefined), A (absolute), T (text segment symbol), D (data segment symbol), B (bss segment symbol), C (common symbol), f file name, or – for dbx(1) symbol table entries (see – **a** below). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

Options are:

– **a** Print dbx debugger symbol table entries.

– **g** Print only global (external) symbols.

– **n** Sort numerically rather than alphabetically.

– **o** Prepend file or archive element name to each output line rather than only once.

– **p** Don't sort; print in symbol-table order.

– **r** Sort in reverse order.

– **u** Print only undefined symbols.

SEE ALSO

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

NAME

nohup – run a command immune to hangups and quits

SYNTAX

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 **nohup.out** in the user's home directory (\$HOME/nohup.out).

EXAMPLE

To execute a command in the background that will not be terminated if the user logs off:

nohup *command* **&**

SEE ALSO

nice(1), signal(2).

NAME

od – octal dump

SYNTAX

od [– **bcdosx**] [file] [[+]offset[.][**b**]]

DESCRIPTION

Od dumps *file* in the format selected by a format option. 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. If no option is given, **this is the default.**

– **s** Interpret 16-bit words in signed decimal.

– **x** Interpret words in hex.

If *file* is omitted, the standard input is read and dumped.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended, the offset is interpreted in decimal. If **b** is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded by +.

Dumping continues until end-of-file.

NAME

pack, pcat, unpack – compress and expand files

SYNTAX

pack [–] name ...

pcat name ...

unpack name ...

DESCRIPTION

Pack attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file *name* is replaced by a packed file *name.z* with the same access modes, access and modified dates, and owner as those of *name* . If *pack* is successful, *name* is removed. Packed files can be restored to their original form by *unpack* or **pcat**.

Pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis.

– is a request to print the number of times each byte is used, its relative frequency, and the code for the byte. Additional uses of – in place of *name* will cause the request 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 8192 bytes, 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. 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

With no arguments, *pcat* unpacks the standard input, and therefore can be used as a filter in a pipeline:

```
pcat < packedfile.z |more
```

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

pagesize – print system page size

SYNTAX

pagesize

DESCRIPTION

Pagesize prints the size of a page of memory in bytes, as returned by *getpagesize*(2). This program is useful in constructing portable shell scripts. On Ridge 32 systems the page size is 4096 bytes.

SEE ALSO

getpagesize(2)

NAME

Pasc – Pascal compiler

SYNTAX

pasc [-L listfile] [-O objfile] file.**p**

DESCRIPTION

Pasc activates the PASCAL compiler. *Pasc* compiles the source **file** and yields an intermediate file called P-code, which can be supplied to the P-code translator *ptrans*(1).

The default name of the output P-code file is **file.q** . **-o** *objfile* overrides the default and specifies the name of the output p-code file.

By default, no output listing is generated. **-l** *listfile* is a request for a listing named *listfile* which contains the compiled source with line numbers and syntax errors marked.

SEE ALSO

cpp(1), pp(1), ptrans(1), link(1)

NAME

passwd – change login password

SYNTAX

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(1), crypt(3C), passwd(4).

NAME

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

SYNTAX

paste file1 file2 ...
paste – dlist file1 file2 ...
paste – s [– dlist] file1 file2 ...

DESCRIPTION

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

pg – file perusal filter for soft-copy terminals

SYNTAX

pg [– *number*] [– **p** *string*] [– **cefns**] [+*linenumber*] [+*/pattern/*] [files...]

DESCRIPTION

The *pg* command is a filter which allows the examination of *files* one screenful at a time on a soft-copy terminal. (The file name – and/or NULL arguments indicate that *pg* should read from the standard input.) Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are enumerated below.

This command is different from previous paginators in that it allows you to back up and review something that has already passed. The method for doing this is explained below.

In order to determine terminal attributes, *pg* scans the *terminfo*(4) data base for the terminal type specified by the environment variable **TERM**. If **TERM** is not defined, the terminal type **dumb** is assumed.

The command line options are:

– *number*
An integer specifying the size (in lines) of the window that *pg* is to use instead of the default. (On a terminal containing 24 lines, the default window size is 23).

– **p** *string*
Causes *pg* to use *string* as the prompt. If the prompt string contains a "%d", the first occurrence of "%d" in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is ":".

– **c** Home the cursor and clear the screen before displaying each page. This option is ignored if **clear_screen** is not defined for this terminal type in the *terminfo*(4) data base.

– **e** Causes *pg* *not* to pause at the end of each file.

– **f** Normally, *pg* splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results. The – *f* option inhibits *pg* from splitting lines.

– **n** Normally, commands must be terminated by a <*newline*> character. This option causes an automatic end of command as soon as a command letter is entered.

– **s** Causes *pg* to print all messages and prompts in standout mode (usually inverse video).

+*linenumber*
Start up at *linenumber*.

+*/pattern/*
Start up at the first line containing the regular expression pattern.

The responses that may be typed when *pg* pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding *address*, an optionally signed number indicating the point from which further text should be displayed. This *address* is interpreted in either pages or lines depending on the command. A signed *address* specifies a point relative to the current page or line, and an unsigned *address* specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are as follows:

(+1)<*newline*> or <*blank*>
This causes one page to be displayed. The address is specified in pages.

(+1) **l** With a relative address this causes *pg* to simulate scrolling the screen, forward or backward, the number of lines specified. With an absolute address this command prints a screenful beginning at the specified line.

(+1) **d** or **^D**
Simulates scrolling half a screen forward or backward.

The following perusal commands take no *address*.

. or **^L** Typing a single period causes the current page of text to be redisplayed.

$ Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available for searching for text patterns in the text. The regular expressions described in *ed*(1) are available. They must always be terminated by a <*newline*>, even if the – *n* option is specified.

i/*pattern*/
Search forward for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

i^*pattern*^
i?*pattern*?
Search backwards for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately before the current page and continues to the beginning of the current file, without wrap-around. The ^ notation is useful for Adds 100 terminals which will not properly handle the ?.

After searching, *pg* will normally display the line found at the top of the screen. This can be modified by appending **m** or **b** to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix **t** can be used to restore the original situation.

The user of *pg* can modify the environment of perusal with the following commands:

***i*n** Begin perusing the *i*th next file in the command line. The *i* is an unsigned number, default value is 1.

***i*p** Begin perusing the *i*th previous file in the command line. *i* is an unsigned number, default is 1.

***i*w** Display another window of text. If *i* is present, set the window size to *i*.

s *filename*
Save the input in the named file. Only the current file being perused is saved. The white space between the **s** and *filename* is optional. This command must always be terminated by a <*newline*>, even if the – *n* option is specified.

h Help by displaying an abbreviated summary of available commands.

q or **Q** Quit *pg*.

!*command*
Command is passed to the shell, whose name is taken from the SHELL environment variable. If this is not available, the default shell is used. This command must always be terminated by a <*newline*>, even if the – *n* option is specified.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\) or the interrupt (break) key. This causes *pg* to stop sending output, and display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

If the standard output is not a terminal, then *pg* acts just like *cat*(1), except that a header is printed before each file (if there is more than one).

EXAMPLE

A sample usage of *pg* in reading system news would be

```
news |pg -p "(Page %d):"
```

NOTES

While waiting for terminal input, *pg* responds to BREAK, DEL, and ^ by terminating execution. Between prompts, however, these signals interrupt *pg*'s current task and place the user in prompt mode. These should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

Users of Berkeley's *more* will find that the z and f commands are available, and that the terminal /, ^, or ? may be omitted from the searching commands.

FILES

/usr/lib/terminfo/*
: Terminal information data base

/tmp/pg*
: Temporary file when input is from a pipe

SEE ALSO

crypt(1), ed(1), grep(1), terminfo(4).

BUGS

If terminal tabs are not set every eight positions, undesirable results may occur.

When using *pg* as a filter with another command that changes the terminal I/O options (e.g., *crypt*(1)), terminal settings may not be restored correctly.

NAME

pp – Pascal invoker

SYNTAX

pp [options] ... file ...

DESCRIPTION

Pp invokes a series of programs that compile and ready a Pascal program for execution.

File names ending with **.p** are taken to be Pascal source programs. They are compiled into object files whose names are the same as the source, but with **.o** substituted for the **.p**. The **.o** file is normally deleted, however, unless an option is specified to request that it be kept.

File names ending with **.q** are assumed to be pasc(1) output files. Each **.q** file is passed through ptrans(1), resulting in an object file with **.o** replacing the **.q**.

Options: See *link(1)* for link editor options and *cpp*(1) for more preprocessor options.

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

– **o***filename*
Name the executable file *filename*, not the default **a.out**.

– **S** Compile the named Pascal programs, and leave the ptrans(1) output on corresponding files suffixed **.q**.

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

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

– **W***c,arg1[,arg2...]*
Hand off the argument[s] *argi* to pass *c* where *c* is one of [**p0al**] indicating preprocessor, compiler, ptrans, or link editor, respectively.

– **m mapfile**
create a load map file called *mapfile*, like that produced by map(1).

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

FILES

file.p	input file
file.o	object file
a.out	linked output
/tmp/ctm*	(a name starting with "ctm") temporary
/lib/cpp	C preprocessor *cpp*(1)
/bin/pasc	Ridge compiler, *pasc(1)*
/bin/ptrans	p-code translator
/bin/link	link editor, *link(1)*
/lib/rtl.o	standard Pascal library

EXAMPLE

```
$ pp pgm.p
$ a.out     /* executes the program */
```

What Happens:

The C preprocessor cpp(1) **/lib/cpp** executes. Its input file is **pgm.p**, and its output is a temporary file called **/tmp/ctmabc**.

The Pascal compiler pasc(1) **/bin/pasc** executes. Its input file is **/tmp/ctmabc** and its output file is **/tmp/ctmdef**.

The p-code translator ptrans(1) **/bin/ptrans** executes. Its input file is **/tmp/ctmdef** and its output file is **/tmp/ctmghi**.

The linker link(1) for Pascal and Ridge assembler is executed. Its input files are **/tmp/ctmghi** and **/lib/rtl.o**. The output file is **a.out**, which is a runnable program.

SEE ALSO

cpp(1), rasm(1), ptrans(1), link(1)

BUGS

Pascal cannot be called from a C or FORTRAN program.

NAME

pr – print files

SYNTAX

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, the date and time of the last file modification, 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.
- **-sc** Separate columns by the single character *c* instead of by the appropriate number of spaces (default for *c* is a tab).

EXAMPLES

Print **file1** and **file2** as a double-spaced, three-column listing headed by "file list":

```
pr -3dh "file list" file1 file2
```

Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, 37, . . . :

```
pr -e9 -t <file1 >file2
```

FILES

/dev/tty*

SEE ALSO

cat(1).

NAME

printenv – print out the environment

SYNTAX

printenv [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a *name* is specified, only its value is printed.

If a *name* is specified and it is not defined in the environment, *printenv* returns exit status 1, else it returns status 0.

SEE ALSO

sh(1), environ(6), csh(1)

NAME

prmail – print out mail in the post office

SYNTAX

prmail [user ...]

DESCRIPTION

Prmail prints the mail which waits for you, or the specified user, in the post office. The mail is not disturbed.

FILES

/usr/mail/* post office

SEE ALSO

mail(1), from(1)

NAME

prs – print an SCCS file

SYNTAX

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
: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: :D

:

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 *ROS Utility Guide*.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ps – report process status

SYNTAX

ps [-als] [-t *terminal* [,*terminal*] ...]

DESCRIPTION

Ps prints the status of processes. By default, **ps** shows the processes associated with the controlling terminal only. The process id, runtime, state, and command name are printed in the default form.

– **l** is a request for the long form, which also displays the PC, parent, number of page faults, priority, and kernel message waiting flag.

– **s** shows status of system processes, like drivers, memory managers, and disc managers.

– **a** shows the status of all processes.

– **t** shows the status of all processes associated with terminals in the list. The list is separated from the option selections by a space. The list elements are spearated by commas or the entire list may be enclosed in quotation marks. Recognizable terminal names are:

nnn	/dev/tty*nnn* (eg: 3 is /dev/tty3)
tty*nn*	/dev/tty*nn*
disp*n*	/dev/disp*n*

where *n* is a digit in the range 0 to 9.
The long output form is:

ProcessId	**PC**	**Parent**	**Runtime**	**Faults**	**Pri**	**State**	**Msg**	**Command**
ProcessId	*PC*	*Parent*	*Runtime*	*Faults*	*Pri*	*State*	*Msg*	*Command*
ProcessId	*PC*	*Parent*	*Runtime*	*Faults*	*Pri*	*State*	*Msg*	*Command*
...	...	...	...	...	...	...	...	...

ProcessId is the unique process identifier.

PC is the current value of the program counter.

Parent is the process ID of this process' parent.

Runtime is the CPU time used by this process, given in the form *hours:minutes:seconds.1000ths-of-a-second.*

Faults is the number of page faults this process has incurred.

Pri is the current execution priority of the process.

State is:

R if the process is runnable (but is not necessarily in the queue for the CPU).
S if the process is suspended.
F if the process is faulted.

Msg is the kernel message waiting flag:
N if no messages are waiting.
Y if a message is waiting.

Command is the command name and first argument(s), or the filename of an open file manager process.

EXAMPLE

ps -lt "disp2,0 tty3"

shows all processes associated with display 2, and terminals tty0 and tty3.

BUGS

The command arguments of PASCAL programs cannot be printed. PASCAL program names are printed only if the given PASCAL program lives in /bin, /usr/bin, or in the current working directory. Open file managers can be found only in **/tmp**, **/usr/tmp**, or the working directory.

SEE ALSO

who(1), nice(1)

FILES

/ros
/drivers
/bin
/usr/bin
/etc/utmp

NAME

ptrans – P-code translator

SYNTAX

ptrans [-L listfile] [-O objfile] file.**q**

DESCRIPTION

Ptrans accepts an input P-code *file*.**q** and translates it into an object file of the same name ending with **.o** . The resulting object file is in "hex" format (representing Ridge machine instructions), and is suitable as input to the object module linker *link*(1).

-o is a request to name the output object file *objfile*. By default, no listing is generated.

-l *listfile* is a request for a listing named *listfile* which contains an assembly language representation of the translated code.

SEE ALSO

link(1), pasc(1), pp(1)

NAME

pwck, grpck – password/group file checkers

SYNTAX

/etc/pwck [file]
/etc/grpck [file]

DESCRIPTION

Pwck scans the password file and notes any inconsistencies. The checks include validation of the number of fields, login name, user ID, group ID, and whether the login directory and optional program name exist. A valid login name is eight characters or less, consists of lowercase letters and digits only, and begins with an alphabetic character, The default password file is **/etc/passwd**.

Grpck verifies all entries in the group file. This verification includes a check of the number of fields, group name, group ID, and whether all login names appear in the password file. The default group file is **/etc/group**.

FILES

/etc/group
/etc/passwd

SEE ALSO

group(4), passwd(4).

DIAGNOSTICS

Group entries in **/etc/group** with no login names are flagged.

NAME

pwd – working directory name

SYNTAX

pwd

DESCRIPTION

Pwd prints the path name of the working (current) directory.

SEE ALSO

cd(1).

NAME

ranlib – convert archives to random libraries

SYNTAX

ranlib archive ...

DESCRIPTION

Ranlib converts each *archive* to a form which the loader can load more rapidly. *Ranlib* does this by adding a table of contents called _.SYMDEF to the beginning of the archive. *Ranlib* uses *ar*(1) to reconstruct the archive, so that sufficient temporary file space must be available in the file system which contains the current directory.

SEE ALSO

ld(1), ar(1), lorder(1)

BUGS

Because generation of a library by *ar* and randomization of the library by *ranlib* are separate processes, phase errors are possible.

NAME

ratfor – rational Fortran dialect

SYNTAX

ratfor [options] [files]

DESCRIPTION

Ratfor converts a rational dialect of Fortran into ordinary irrational Fortran. *Ratfor* provides control flow constructs essentially identical to those in C:

statement grouping:

{ statement; statement; statement }

decision-making:

if (condition) statement [**else** statement]
switch (integer value) {
    case integer: statement
    ...
    [**default**:] statement
}

loops:

while (condition) statement
for (expression; condition; expression) statement
do limits statement
repeat statement [**until** (condition)]
break
next

and other tools to make FORTRAN programming easier:

free form input:

multiple statements per line; automatic continuation

comments:

this is a comment.

translation of relationals:

>, >=, etc., become **.GT.**, **.GE.**, etc.

return expression to caller from function:

return (expression)

define:

define *name replacement*

include:

include *file*

The option **–h** causes quoted strings to be turned into **27H** constructs. The **–C** option copies comments to the output and attempts to format it neatly. Normally, continuation lines are marked with a **&** in column 1; the option **–6x** makes the continuation character **x** and places it in column 6.

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

SEE ALSO

f77(1)

NAME

rcp – remote file copy

SYNTAX

rcp file1 file2
rcp [**–r**] file ... directory

DESCRIPTION

Rcp copies files between machines. Each *file* or *directory* argument is either a remote file name of the form "rhost:path", or a local file name (containing no ':' characters, or a '/' before any ':'s.)

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

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

Rcp does not prompt for passwords; your current local user name must exist on *rhost* and allow remote command execution via *rsh*(1).

Rcp handles third party copies, where neither source nor target files are on the current machine. Hostnames may also take the form "rhost.rname" to use *rname* rather than the current user name on the remote host.

SEE ALSO

ftp(1), rsh(1), rlogin(1)

BUGS

Doesn't detect all cases where the target of a copy might be a file in cases where only a directory should be legal.
Is confused by any output generated by commands in a *.login*, *.profile*, or *.cshrc* file on the remote host.

If the source and destination directories are local, *rcp* calls *cp*(1) to do the local copy. *cp*(1) does not accept the **-r** arguement.

The usage error message is not printed if a typing error was made.

NAME

redit – display-oriented text editor

SYNTAX

redit [file]

DESCRIPTION

Redit is a text editor designed for use with the Ridge Graphics Display and Ridge Text Terminal, utilizing their keyboard function keys and full screen editing capabilities. For full documentation, see the Ridge Text Editor Reference Manual.

If <file> is specified, it is made available for editing immediately after the editor is activated. Otherwise, the editor is activated but no particular file is opened for editing.

SEE ALSO

Ridge Text Editor Reference Manual

NAME

redraw – repaint entire display surface

SYNTAX

redraw

DESCRIPTION

Redraw is used to repaint the entire display surface, including the background and all windows, on the Ridge display *disp*(7).

Repainting is usually only necessary after running a program which writes directly to the display, rather than using the display management software, in order to write over parts of windows.

SEE ALSO

Ridge Multi-Window Display Management Guide
copybits(3), graf(3), windows(3), disp(7)

NAME

regcmp – regular expression compile

SYNTAX

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

rev – reverse lines of a file

SYNTAX

rev [file] ...

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line.

If no file is specified, the standard input is copied.

NAME

rlogin – remote login

SYNTAX

rlogin rhost [**–e** *c*] [**–l** username]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* to the remote host system *rhost.*

Each host has a file */etc/hosts.equiv* which contains a list of *rhost*'s with which it shares account names. (The host names must be the standard names as described in *rsh*(1).) When you *rlogin* as the same user on an equivalent host, you don't need to give a password. Each user may also have a private equivalence list in a *.rhosts* file in his login directory. Each line in this file should contain an *rhost* and a *username* separated by a space, giving additional cases where logins without passwords are to be permitted. If the originating user is not equivalent to the remote user, then a login and password will be prompted for on the remote machine as in *login*(1). To avoid some security problems, the *.rhosts* file must be owned by either the remote user or root and may not be a symbolic link.

All echoing takes place at the remote site, so that (except for delays) the rlogin is transparent. Flow control via ^S and ^Q and flushing of input and output on interrupts are handled properly. A line of the form "~." disconnects from the remote host, where "~" is the escape character. A different escape character may be specified by the **–e** option. There is no space separating this option flag and the argument character.

SEE ALSO

rsh(1)

NAME

rlogind – remote login server

SYNTAX

/etc/rlogind [**–d**]

DESCRIPTION

Rlogind is the server for the *rlogin*(1) program. The server provides a remote login facility with authentication based on privileged port numbers.

Rlogind listens for service requests at the port indicated in the "login" service specification; see *services*(4). When a service request is received, the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts*(4)), the server aborts the connection.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal (see *pty*(7)), and manipulates file descriptors so that the slave half of the pseudo terminal becomes the **stdin**, **stdout**, and **stderr** for a login process. The login process is an instance of the *login*(1) program, invoked with the **–R** option. The login process then proceeds with the authentication process as described in *rshd*(1), but if automatic authentication fails, it reprompts the user to login as one finds on a standard terminal line.

The parent of the login process manipulates the master side of the pseudo terminal, operating as an intermediary between the login process and the client instance of the *rlogin* program. In normal operation, the packet protocol described in *pty*(4) is invoked to provide ^S/^Q type facilities and propagate interrupt signals to the remote programs.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1.

"Hostname for your address unknown."
No entry in the host name database existed for the client's machine.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

NAME

rm, rmdir – remove files or directories

SYNTAX

rm [– **fri**] file ...

rmdir dir ...

DESCRIPTION

Rmdir removes one or more directories, which must be empty.

Rm removes one or more file entries 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 *file* has no write permission, and the standard input is the terminal, **rm** prints its access code and prompts the user for go-ahead to remove that file. User response must begin with **y**, or the file will not be removed.

-f specifies that no questions will be asked before **rm** removes files without write permission.

If the standard input is not a terminal, no questions are asked.

Rm -r *directory* causes **rm** to remove each file in the directory (unless it has no write permission in which case it will ask), and then the directory itself.

Rm -i *files* specifies that **rm** will interactively ask whether to delete each file.

Rm -ir *directories* specifies that **rm** will ask which named directories you want to examine, and then which files within your selected directories you want to remove, then which directories you want to remove.

It is forbidden to remove the file .. merely to avoid the antisocial consequences of inadvertently doing something like:

rm – r .*

SEE ALSO

unlink(2).

NAME

rmail – handle remote mail received via uucp

SYNTAX

rmail user ...

DESCRIPTION

Rmail interprets incoming mail received via *uucp*(1), collapsing "From" lines in the form generated by *binmail*(1) into a single line of the form "return-path!sender", and passing the processed mail on to *sendmail*(1).

Rmail is explicitly designed for use with *uucp* and *sendmail*.

SEE ALSO

binmail(1), uucp(1), sendmail(1)

BUGS

Rmail hangs if executed directly.

NAME

rmdel – remove a delta from an SCCS file

SYNTAX

rmdel – r*SID* files

DESCRIPTION

Rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the specified must *not* be that of a version being edited for the purpose of making a delta (i. e., if a *p-file* (see *get*(1)) exists for the named SCCS file, the specified must *not* appear in any entry of the *p-file*).

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 *Interactive Access to ROS.*

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

rsh – remote shell

SYNTAX

rsh host [–**l** username] [–**n**] command

DESCRIPTION

Rsh connects to the specified *host,* and executes the specified *command. Rsh* copies its standard input to the remote command, the standard output of the remote command to its standard output, and the standard error of the remote command to its standard error. Interrupt, quit and terminate signals are propagated to the remote command; *rsh* normally terminates when the remote command does.

The remote username used is the same as your local username, unless you specify a different remote name with the –**l** option. This remote name must be equivalent (in the sense of *rlogin*(1)) to the originating account; no provision is made for specifying a password with a command.

If you omit *command,* then, instead of executing a single command, you will be logged in on the remote host using *rlogin*(1).

Shell metacharacters which are not quoted are interpreted on local machine, while quoted metacharacters are interpreted on the remote machine. Thus, the command

```
rsh otherhost cat remotefile >> localfile
```

appends the remote file *remotefile* to the localfile *localfile,* while

```
rsh otherhost cat remotefile ">>" otherremotefile
```

appends *remotefile* to *otherremotefile.*

Host names are given in the file /etc/hosts. Each host has one standard name (the first name given in the file) which is rather long and unambiguous, and optionally one or more nicknames.

FILES

/etc/hosts

SEE ALSO

rlogin(1)

BUGS

If you are using *csh*(1) and you put a *rsh*(1) in the background without redirecting its input away from the terminal, it will block even if no reads are posted by the remote command. If no input is desired, you should redirect the input of *rsh* to /dev/null using the –**n** option.

You cannot run an interactive command (like *vi*(1)). Instead, use *rlogin*(1).

Stop signals terminate the local *rsh* process only. This is arguably wrong, but currently hard to fix for reasons too complicated to explain here.

Remote commands that treat standard out and standard in do not work as usual because standard out is not a terminal for the remote shell. For instance, **ll -C** must be used to force the long listing into a columns.

NAME

rshd – remote shell server

SYNTAX

/etc/rshd

DESCRIPTION

Rshd is the server for the *rsh*(1) program. The server provides remote execution facilities with authentication based on privileged port numbers.

Rshd listens for service requests at the port indicated in the "cmd" service specification; see *services*(4). When a service request is received, the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server reads characters from the socket up to a null ('\0') byte. The resulting string is interpreted as an ASCII number, base 10.

3) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine. The source port of this second connection is also in the range 0-1023.

4) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts*(4)), the server aborts the connection.

5) A null terminated user name of, at most, 16 characters is retrieved on the initial socket. This user name is interpreted as a user identity to use on the **server**'s machine.

6) A null terminated user name of, at most, 16 characters is retrieved on the initial socket. This user name is interpreted as the user identity on the **client**'s machine.

7) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

8) *Rshd* then validates the user according to the following steps. The remote user name is looked up in the password file and a *chdir* is performed to the user's home directory. If either the lookup or *chdir* fail, the connection is terminated. If the user is not the super-user, (user id 0), the file */etc/hosts.equiv* is consulted for a list of hosts considered "equivalent". If the client's host name is present in this file, the authentication is considered successful. If the lookup fails, or the user is the super-user, then the file *.rhosts* in the home directory of the remote user is checked for the machine name and identity of the user on the client's machine. If this lookup fails, the connection is terminated.

9) A null byte is returned on the connection associated with the **stderr** and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rshd*.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 9 above upon successful completion of all the steps prior to the command execution).

"locuser too long"
The name of the user on the client's machine is longer than 16 characters.

"remuser too long"
The name of the user on the remote machine is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Hostname for your address unknown."
No entry in the host name database existed for the client's machine.

"Login incorrect."
No password file entry for the user name existed.

"No remote directory."
The *chdir* command to the home directory failed.

"Permission denied."
The authentication procedure described above failed.

"Can't make pipe."
The pipe needed for the **stderr**, wasn't created.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

SEE ALSO
rsh(1)

BUGS
The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

NAME

ruptime – show host status of local machines

SYNTAX

ruptime [**–a**] [**–l**] [**–t**] [**–u**]

DESCRIPTION

Ruptime gives a status line like *uptime* for each machine on the local network; these are formed from packets broadcast by each host on the network once a minute.

Machines for which no status report has been received for 5 minutes are shown as being down.

Users idle an hour or more are not counted unless the **–a** flag is given.

Normally, the listing is sorted by host name. The **–l** , **–t** , and **–u** flags specify sorting by load average, uptime, and number of users, respectively.

FILES

/usr/spool/rwho/whod.* data files

SEE ALSO

rwho(1)

NAME

rwho – who's logged in on local machines

SYNTAX

rwho [**–a**]

DESCRIPTION

The *rwho* command produces output similar to *who,* but for all machines on the local network. If no report has been received from a machine for 5 minutes, then *rwho* assumes the machine is down and does not report users last known to be logged into that machine.

If a user hasn't typed to the system for a minute or more, then *rwho* reports this idle time. If a user hasn't typed to the system for an hour or more, then the user will be omitted from the output of *rwho* unless the **–a** flag is given.

FILES

/usr/spool/rwho/whod.* information about other machines

SEE ALSO

ruptime(1), rwhod(1)

BUGS

This is unwieldy when the number of machines on the local net is large.

Idle time is not reported.

NAME

rwhod – system status server

SYNTAX

/etc/rwhod

DESCRIPTION

Rwhod is the server which maintains the database used by the *rwho*(1) and *ruptime*(1) programs. Its operation is predicated on the ability to *broadcast* messages on a network.

Rwhod operates as both a producer and consumer of status information. As a producer of information, it periodically queries the state of the system and constructs status messages which are broadcast on a network. As a consumer of information, it listens for other *rwhod* servers' status messages, validating them, then recording them in a collection of files located in the directory */usr/spool/rwho*.

The *rwho* server transmits and receives messages at the port indicated in the "rwho" service specification, see *services*(4). The messages sent and received are of the form:

```
struct  outmp {
        char    out_line[8];/* tty name */
        char    out_name[8];/* user id */
        long    out_time;/* time on */
};

struct  whod {
        char    wd_vers;
        char    wd_type;
        char    wd_fill[2];
        int     wd_sendtime;
        int     wd_recvtime;
        char    wd_hostname[32];
        int     wd_loadav[3];
        int     wd_boottime;
        struct  whoent {
                struct  outmp we_utmp;
                int     we_idle;
        } wd_we[1024 / sizeof (struct whoent)];
};
```

The array at the end of the message contains information about the users logged in to the sending machine. This information includes the contents of the *utmp*(4) entry for each non-idle terminal line and a value indicating the time since a character was last received on the terminal line.

Messages received by the *rwho* server are discarded unless they originated at a *rwho* server's port. In addition, if the host's name in the message contains any unprintable ASCII characters, the message is discarded. Valid messages received by *rwhod* are placed in files named *whod.hostname* in the directory */usr/spool/rwho*. These files contain only the most recent message, in the format described above.

Status messages are generated approximately once every 60 seconds.

SEE ALSO

rwho(1), ruptime(1)

BUGS

Should relay status information between networks. People often interpret the server dying as a machine going down.

NAME

sact – print current SCCS file editing activity

SYNTAX

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

SYNTAX

sccsdiff –**r**SID1 –**r**SID2 [–**p**] [–**s**n] files

DESCRIPTION

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

–**r***SID?*	*SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to *bdiff*(1) in the order given.
–**p**	pipe output for each file through *pr*(1).
–**s***n*	*n* is the file segment size that *bdiff* will pass to *diff*(1). This is useful when *diff* fails due to a high system load.

FILES

/tmp/get????? Temporary files

SEE ALSO

bdiff(1), get(1), help(1), pr(1).
Source Code Control System in *ROS Utility Guide*.

DIAGNOSTICS

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

NAME

sdiff – side-by-side difference program

SYNTAX

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

SYNTAX

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 list that follows, the maximum allowable number of 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 end of 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; long lines are folded.

(2) **n** Copy the pattern space to the standard output. Replace pattern space with next input line.

(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 pattern space to 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 end of script.

(2) **w** *wfile* Write. Append the pattern space to *wfile*.

(2) **x** Exchange the contents of 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

sendmail – send mail over the internet

SYNTAX

/usr/lib/sendmail [flags] [address ...]

newaliases

mailq

DESCRIPTION

Sendmail sends a message to one or more people, routing the message over whatever networks are necessary. *Sendmail* does internetwork forwarding as necessary to deliver the message to the correct place.

Sendmail is not intended as a user interface routine. Other programs provide user-friendly front ends, while *sendmail* is used only to deliver pre-formatted messages.

With no flags, *sendmail* reads its standard input up to a control-D or a line with a single dot and sends a copy of the letter found there to all of the addresses listed. It determines the network to use based on the syntax and contents of the addresses.

Local addresses are looked up in a file and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash. Normally, the sender is not included in any alias expansions (e.g., if 'john' sends to 'group' and 'group' includes 'john' in the expansion) then the letter will not be delivered to 'john'.

Flags are:

–ba	Go into ARPANET mode. All input lines must end with a CR-LF, and all messages will be generated with a CR-LF at the end. Also, the "From:" and "Sender:" fields are examined for the name of the sender.
–bd	Run as a daemon.
–bi	Initialize the alias database.
–bm	Deliver mail in the usual way (default).
–bp	Print a listing of the queue.
–bs	Use the SMTP protocol as described in RFC821. This flag implies all the operations of the **–ba** flag that are compatible with SMTP.
–bt	Run in address test mode. This mode reads addresses and shows the steps in parsing; it is used for debugging configuration tables.
–bv	Verify names only – do not try to collect or deliver a message. Verify mode is normally used for validating users or mailing lists.
–bz	Create the configuration freeze file.
–C*file*	Use alternate configuration file.
–d*X*	Set debugging value to *X*.
–F*fullname*	Set the full name of the sender.
–f*name*	Sets the name of the "from" person (i.e., the sender of the mail). **–f** can only be used by the special users *root, daemon,* and *network,* or if the person you are trying to become is the same as the person you are.
–h*N*	Set the hop count to *N*. The hop count is incremented every time the mail is processed. When it reaches a limit, the mail is returned with an error message, the victim of an aliasing loop.

−n	Don't do aliasing.
−o*x value*	Set option *x* to the specified *value*. Options are described below.
−q[*time*]	Process saved messages in the queue at given intervals. If *time* is omitted, process the queue once. *Time* is given as a tagged number, with 's' being seconds, 'm' being minutes, 'h' being hours, 'd' being days, and 'w' being weeks. For example, "−q1h30m" or "−q90m" would both set the timeout to one hour thirty minutes.
−r*name*	An alternate and obsolete form of the **−f** flag.
−t	Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for people to send to. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed.
−v	Go into verbose mode. Alias expansions will be announced, etc.

There are also a number of processing options that may be set. Normally these will only be used by a system administrator. Options may be set either on the command line using the **−o** flag or in the configuration file. These are described in detail in the *Installation and Operation Guide*. The options are:

A*file*	Use alternate alias file.
c	On mailers that are considered "expensive" to connect to, don't initiate immediate connection. This requires queueing.
d*x*	Set the delivery mode to *x*. Delivery modes are 'i' for interactive (synchronous) delivery, 'b' for background (asynchronous) delivery, and 'q' for queue only (i.e., actual delivery is done the next time the queue is run).
D	Try to automatically rebuild the alias database, if necessary.
e*x*	Set error processing to mode *x*. Valid modes are 'm' to mail back the error message, 'w' to "write" back the error message (or mail it back if the sender is not logged in), 'p' to print the errors on the terminal (default), 'q' to throw away error messages (only exit status is returned), and 'e' to do special processing for the BerkNet. If the text of the message is not mailed back by modes 'm' or 'w' and if the sender is local to this machine, a copy of the message is appended to the file "dead.letter" in the sender's home directory.
F*mode*	The mode to use when creating temporary files.
f	Save UNIX-style From lines at the front of messages.
g*N*	The default group id to use when calling mailers.
H*file*	The SMTP help file.
i	Do not take dots on a line by themselves as a message terminator.
L*n*	The log level.
m	Send to "me" (the sender) if I am in an alias expansion.
o	If set, this message may have old style headers. If not set, this message is guaranteed to have new style headers (i.e., commas instead of spaces between addresses). If set, an adaptive algorithm is used that will correctly determine the header format in most cases.
Q*queuedir*	Select the directory in which to queue messages.
r*timeout*	The timeout on reads; if none is set, *sendmail* will wait forever for a mailer.

S*file*	Save statistics in the named file.
s	Always instantiate the queue file, even under circumstances where it is not strictly necessary.
T*time*	Set the timeout on messages in the queue to the specified time. After sitting in the queue for this amount of time, they will be returned to the sender. The default is three days.
t*stz,dtz*	Set the name of the time zone.
u*N*	Set the default user id for mailers.

If the first character of the user name is a vertical bar, the rest of the user name is used as the name of a program to pipe the mail to. It may be necessary to quote the name of the user to keep *sendmail* from suppressing the blanks from between arguments.

Sendmail returns an exit status describing what it did. The codes are defined in *<sysexits.h>*

EX_OK	Successful completion on all addresses.
EX_NOUSER	User name not recognized.
EX_UNAVAILABLE	Catchall, meaning necessary resources were not available.
EX_SYNTAX	Syntax error in address.
EX_SOFTWARE	Internal software error, including bad arguments.
EX_OSERR	Temporary operating system error, such as cannot fork.
EX_NOHOST	Host name not recognized.
EX_TEMPFAIL	Message could not be sent immediately, but was queued.

If invoked as *newaliases, sendmail* will rebuild the alias database. If invoked as *mailq, sendmail* will print the contents of the mail queue.

FILES

Except for */usr/lib/sendmail.cf,* these pathnames are all specified in */usr/lib/sendmail.cf.* Thus, these values are only approximations.

/usr/lib/aliases	raw data for alias names
/usr/lib/aliases.pag	data base of alias names
/usr/lib/aliases.dir	data base of alias names
/usr/lib/sendmail.cf	configuration file
/usr/lib/sendmail.fc	frozen configuration
/usr/lib/sendmail.hf	help file
/usr/lib/sendmail.st	collected statistics
/usr/bin/uux	to deliver uucp mail
/bin/binmail	to deliver local mail
/usr/spool/mqueue/*	temp files

SEE ALSO

binmail(1), mail(1), aliases(4), rmail(1), mailaddr(5);
DARPA Internet Request For Comments RFC819, RFC821, RFC822;
Sendmail Installation and Operation Guide in the Ridge Utility Guide.

BUGS

Sendmail converts blanks in addresses to dots. This is incorrect according to the old ARPANET mail protocol RFC733 (NIC 41952), but is consistent with the new protocols (RFC822).

When used with csh "!" must be backslashed.

NAME

setfont – set the font in a window

SYNTAX

setfont [file]

DESCRIPTION

Setfont is used to determine or modify the font associated with a window on the Ridge Display *disp*(7). *File* is the pathname of a font specification file.

Standard font files are held in the **/fonts** directory. The format of the bit-matrix font file is described in font(4).

If no *file* is given, *setfont* displays the name of the font in use within the window in which it is executed.

Each window may have its own font.

FILES

/fonts/*

SEE ALSO

windows(3X), font(4)

NAME

setmnt – establish mount table

SYNTAX

/etc/setmnt

DESCRIPTION

Setmnt creates the **/etc/mnttab** table (see *mnttab*(4)), which is needed for both the *mount*(1) and *umount* commands. *Setmnt* reads standard input and creates a *mnttab* entry for each line. Input lines have the format:

filesys node

where *filesys* is the name of the file system's *special file* (e.g., "rp??") and *node* is the root name of that file system. Thus *filesys* and *node* become the first two strings in the *mnttab(4)* entry.

FILES

/etc/mnttab

SEE ALSO

mnttab(4).

BUGS

Evil things will happen if *filesys* or *node* are longer than 10 characters.
Setmnt silently enforces an upper limit on the maximum number of *mnttab* entries.

NAME

settek – set a window to Tektronix 4014 emulation mode

SYNTAX

settek

DESCRIPTION

Settek activates Tektronix 4014[tm] emulation mode in the Ridge Monochrome Display window in which it is executed. The window is initially cleared, Alpha mode is set, and the cursor is set at the home position.

Tektronix 4014 mode can also be set by typing **ESC %! 0** .

Tektronix 4014 mode remains in effect until X3.64 mode is activated with setx3.64(1) or by typing **ESC %! 1** .

SEE ALSO

setx3.64(1), wgraf(3X), windows(3X), disp(7)

NAME

settitle – set the title of a window

SYNTAX

settitle [*name*]

DESCRIPTION

Settitle is used to determine or modify the title associated with a window on the Ridge display *disp*(7). If an argument is given, *settitle* sets the string that is used when displaying the title tab for the window to *name*.

If no argument is given, the title string associated with the window where *settitle* is run will be printed on the standard output.

SEE ALSO

windows(3)

NAME

setx3.64 – set a window to ANSI X3.64 emulation mode

SYNTAX

setx3.64

DESCRIPTION

Setx3.64 activates ANSI X3.64 emulation mode in the Ridge Monochrome Display window in which it is executed.

ANSI X3.64 mode can also be set by typing **ESC %! 1** .

ANSI X3.64 mode remains in effect until Tektronix 4014^{tm} mode is activated with settek(1) or by typing **ESC %! 0** . ANSI X3.64 mode is the default terminal emulation mode for a newly opened window.

NOTES

ANSI X3.64 is the American National Standards Institute code number for terminal characteristics, such as implemented by the DEC VT-100 terminal.

SEE ALSO

settek(1), windows(3X), disp(7)

NAME

setwsize – set the size of a window

SYNTAX

setwsize [*width height*]

DESCRIPTION

Setwsize is used to determine or modify the size of a window on the Ridge display *disp*(7). If two decimal numeric arguments are given, *setwsize* sets the size of the window in which it is executed to *width* and *height*.

The size unit depends on the mode of the window. If the window is in ASCII or ANSI X3.64 emulation modes, the width and height are specified in columns and lines of text. Otherwise, the width and height are specified in display coordinates (pixels).

If no argument is given, the width and height (in decimal, separated by a space) of the window where *setwsize* is run will be printed on the standard output. This output string is in suitable format for use as the arguments to a later *setwsize*.

SEE ALSO

settek(1), setx3.64(1), windows(3)

NAME

sh, rshell – shell, the standard/restricted command programming language

SYNTAX

sh [**–acefhiknrstuvx**] [args]
rshell [**–acefhiknrstuvx**] [args]

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. *Rshell* is a restricted version of the standard command interpreter *sh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See *Invocation* below for the meaning of arguments to the shell.

Definitions

A *blank* is a tab or a space. A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a name, a digit, or any of the characters *, @, #, ?, –, $, and !.

Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec*(2)). The *value* of a simple-command is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by | (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. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ; or &. Of these four symbols, ; and & have equal precedence, which is lower than that of && and ||. The symbols && and || also have equal precedence. A semicolon (;) causes sequential execution of the preceding pipeline; an ampersand (&) causes asynchronous execution of the preceding pipeline (i.e., the shell does *not* wait for that pipeline to finish). The symbol && (||) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of 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*) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is

executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns a zero exit status.

while *list* **do** *list* **done**

A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

(*list*)

Execute *list* in a sub-shell.

{*list*;}

list is simply executed.

name () {*list*;}

Define a function which is referenced by *name*. The body of the function is the *list* of commands between { and }. Execution of functions is described below (see *Execution*).

The following words are only recognized as the first word of a command and when not quoted:

if then else elif fi case esac for while until do done { }

Comments

A word beginning with # causes that word and all the following characters up to a 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*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Positional parameters may be assigned values by **set**. Keyword parameters (also known as variables) may be assigned values by writing:

name=*value* [*name*=*value*] ...

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

${*parameter***}**

The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is * or @, all the positional parameters, starting with **$1**, are substituted (separated by spaces). Parameter **$0** is set from argument zero when the shell is invoked.

${*parameter***:-***word***}**

If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*.

${*parameter***:=***word***}**

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

${*parameter***:?***word***}**

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

${*parameter***:+***word***}**

If *parameter* is set and is non-null, substitute *word*; otherwise substitute nothing.

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

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

If the colon (:) is omitted from the above expressions, the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

- **#** The number of positional parameters in decimal.
- **-** Flags supplied to the shell on invocation or by the **set** command.
- **?** The decimal value returned by the last synchronously executed command.
- **$** The process number of this shell.
- **!** The process number of the last background command invoked.

The following parameters are used by the shell:

- HOME The default argument (home directory) for the *cd* command.
- PATH The search path for commands (see *Execution* below). The user may not change PATH if executing under *rshell*.
- CDPATH The search path for the *cd* command.
- MAIL If this parameter is set to the name of a mail file *and* the MAILPATH parameter is not set, the shell informs the user of the arrival of mail in the specified file.
- MAILCHECK This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the MAILPATH or MAIL parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.
- MAILPATH A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is *you have mail*.
- PS1 Primary prompt string, by default "$ ".
- PS2 Secondary prompt string, by default "> ".
- IFS Internal field separators, normally **space**, **tab**, and **new-line**.
- SHELL When the shell is invoked, it scans the environment (see *Environment* below) for this name. If it is found and there is an 'r' in the file name part of its value, the shell becomes a restricted shell.

The shell gives default values to PATH, PS1, PS2, MAILCHECK and IFS. HOME and MAIL are set by *login*(1).

Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (those found in IFS) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

File Name Generation

Following substitution, each command *word* is scanned for the characters *, ?, and [. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character . at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

- * Matches any string, including the null string.
- ? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening ``['' is a "!" any character not enclosed is matched.

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

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

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a ****. 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, the secondary prompt (i.e., the value of **PS2**) is issued.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are *not* passed on to the invoked command; substitution occurs before *word* or *digit* is used:

<word Use file *word* as standard input (file descriptor 0).

>word Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.

>>word Use file *word* as standard output. If the file exists output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.

≪[–]word The shell input is read up to a line that is the same as *word*, or to an end-of-file. The resulting document becomes the standard input. If any character of *word* is quoted, 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 ≪, all leading tabs are stripped from *word* and from the document.

<&digit Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using **>&digit**.

<&– The standard input is closed. Similarly for the standard output using **>&–**.

If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

... 2>&1

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

... 1>*xxx* 2>&1

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e. *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

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

Redirection of output is not allowed in the restricted shell.

Environment

The *environment* (see *environ*(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment (see also **set -a**). A parameter may be removed from the environment with the **unset**command. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by **unset**, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. Thus:

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

are equivalent (as far as the execution of *cmd* is concerned).

If the **−k** flag is set, *all* keyword arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and **c**:

```
echo a=b c
set −k
echo a=b c
```

Signals

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent, with the exception of signal 11 (but see also the **trap** command below).

Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the *Special Commands* listed below, it is executed in the shell process. If the command name does not match a *Special Command*, but matches the name of a defined function, the function is executed in the shell process (note how this differs from the execution of shell procedures). The positional parameters **$1**, **$2**, ... are set to the arguments of the function. If the command name matches neither a *Special Command* nor the name of a defined function, a new process is created and an attempt is made to execute the command via *exec*(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (**:**). The default path is **:/bin:/usr/bin** (specifying the current directory, **/bin**, and **/usr/bin**, in that order). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a **/** the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. A parenthesized command is also executed in a sub-shell.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *execs* later). If the command was found in a relative directory, its

location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the PATH variable is changed or the **hash -r** command is executed (see below).

Special Commands

Input/output redirection is now permitted for these commands. File descriptor 1 is the default output location.

: No effect; the command does nothing. A zero exit code is returned.

. *file* Read and execute commands from *file* and return. The search path specified by **PATH** is used to find the directory containing *file*.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter **HOME** is the default *arg*. The shell parameter CDPATH defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is **<null>** (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a **/** the search path is not used. Otherwise, each directory in the path is searched for *arg*. The *cd* command may not be executed by *rshell*.

echo [*arg* ...]
Echo arguments. See *echo*(1) for usage and description.

eval [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*]
Causes a shell to exit with the exit status specified by *n*. If *n* is omitted the exit status is that of the last command executed (an end-of-file will also cause the shell to exit.)

export [*name* ...]
The given *names* are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, a list of all names that are exported in this shell is printed. Function names may *not* be exported.

hash [**–r**] [*name* ...]
For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. There are certain situations which require that the stored location of a command be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

newgrp [*arg* ...]
Equivalent to **exec newgrp** *arg* .... See *newgrp*(1) for usage and description.

pwd Print the current working directory. See *pwd*(1) for usage and description.

read [*name* ...]
One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. The return code is 0 unless an end-of-file is encountered.

readonly [*name* ...]
The given *names* are marked *readonly* and the values of the these *names* may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed.

return [*n*]
Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

set [−−**aefhkntuvx** [*arg* ...]]
- **−a** Mark variables which are modified or created for export.
- **−e** Exit immediately if a command exits with a non-zero exit status.
- **−f** Disable file name generation
- **−h** Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).
- **−k** All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- **−n** Read commands but do not execute them.
- **−t** Exit after reading and executing one command.
- **−u** Treat unset variables as an error when substituting.
- **−v** Print shell input lines as they are read.
- **−x** Print commands and their arguments as they are executed.
- **−−** Do not change any of the flags; useful in setting **$1** to −.

Using + rather than − causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in **$−**. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given the values of all names are printed.

shift [*n*]
The positional parameters from **$n+1** ... are renamed **$1** .... If *n* is not given, it is assumed to be 1.

test
Evaluate conditional expressions. See *test*(1) for usage and description.

times
Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...
The command *arg* is to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *arg* is absent all trap(s) *n* are reset to their original values. If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *arg* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

type [*name* ...]
For each *name*, indicate how it would be interpreted if used as a command name.

ulimit [**−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.

unset [*name* ...]
For each *name*, remove the corresponding variable or function. The variables PATH, PS1, PS2, MAILCHECK and IFS cannot be unset.

wait [*n*]
Wait for the specified process and report its termination status. If *n* is not given all currently active child processes are waited for and the return code is zero.

Invocation

If the shell is invoked through *exec*(2) and the first character of argument zero is −, commands are initially read from **/etc/profile** and from **$HOME/.profile**, if such files exist. Thereafter, commands are read as described below, which is also the case when the shell is invoked as **/bin/sh**. The flags below are interpreted by the shell on invocation only; Note that unless the **−c** or **−s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

−c *string* If the **−c** flag is present commands are read from *string*.

−s If the **−s** flag is present or if no arguments remain commands are read from the standard input. Any remaining arguments specify the positional parameters. Shell output (except for *Special Commands*) is written to file descriptor 2.

−i If the **−i** flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.

−r If the **−r** flag is present the shell is a restricted shell.

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

Rshell Only

Rshell is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of *rshell* 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, *rshell* 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 *rshell*. Some systems also provide a restricted editor *red*.

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above).

FILES

/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

acctcom(1), cd(1), echo(1), env(1), login(1), newgrp(1), pwd(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).

CAVEATS

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the **hash** command to correct this situation.

If you move the current directory or one above it, **pwd** may not give the correct response. Use the **cd** command with a full path name to correct this situation.

NAME

shutdown – terminate all processing

SYNTAX

/etc/shutdown

DESCRIPTION

Shutdown is part of the ROS System operation procedures. Its primary function is to terminate all currently running processes in an orderly and cautious manner. The procedure is designed to interact with the operator (i.e., the person who invoked *shutdown*). *Shutdown* may instruct the operator to perform some specific tasks, or to supply certain responses before execution can resume. *Shutdown* goes through the following steps:

All users logged on the system are notified to log off the system by a broadcasted message. The operator may display his/her own message at this time. Otherwise, the standard file save message is displayed.

If the operator wishes to run the file-save procedure, *shutdown* unmounts all file systems.

All file systems' super blocks are updated before the system is to be stopped (see *sync*(1)). This must be done before re-booting the system, to insure file system integrity. The most common error diagnostic that will occur is *device busy*. This diagnostic happens when a particular file system could not be unmounted.

SEE ALSO

NAME

size – size of an object file

SYNTAX

size [object ...]

DESCRIPTION

Size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in hex and decimal, of each object-file argument. If no file is specified, **a.out** is used.

SEE ALSO

a.out(5)

NAME

sleep – suspend execution for an interval

SYNTAX

sleep time

DESCRIPTION

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

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

or to execute a command every so often:

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

Time is a virtually unlimited integer.

SEE ALSO

alarm(2), sleep(3C).

NAME

sno – SNOBOL interpreter

SYNTAX

sno [files]

DESCRIPTION

Sno is a SNOBOL3 compiler and interpreter (with slight differences). *Sno* obtains input from the concatenation of the named *files* and the standard input. All input through a statement containing the label **end** is considered program and is compiled. The rest is available to **syspit**.

Sno differs from SNOBOL in the following ways:

There are no unanchored searches. To get the same effect:

```
a ** b          unanchored search for b.
a *x* b = x c   unanchored assignment
```

There is no back referencing.

```
x = "abc"
a *x* x         is an unanchored search for abc.
```

Function declaration is done at compile time by the use of the (non-unique) label **define**. Execution of a function call begins at the statement following the **define**. Functions cannot be defined at run time, and the use of the name **define** is preempted. There is no provision for automatic variables other than parameters. Examples:

```
define f( )
define f(a, b, c)
```

All labels except **define** (even **end**) must have a non-empty statement.

Labels, functions and variables must all have distinct names. In particular, the non-empty statement on **end** cannot merely name a label.

If **start** is a label in the program, program execution will start there. If not, execution begins with the first executable statement; **define** is not an executable statement.

There are no built-in functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators / and * must be set off by spaces.

The right side of assignments must be non-empty.

Either ' or " may be used for literal quotes.

The pseudo-variable **sysppt** is not available.

SEE ALSO

awk(1).

NAME

sort – sort and/or merge files

SYNTAX

sort [– **cmu**] [– **o**output] [– **y**kmem] [– **z**recsz] [– **dfiMnr**] [– **btx**] [+pos1 [– pos2]] [files]

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The standard input is read if – is used as a file name or no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

The following options alter the default behavior:

– **c** Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

– **m** Merge only, the input files are already sorted.

– **u** Unique: suppress all but one in each set of lines having equal keys.

– **o***output*

The argument given is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. There may be optional blanks between – **o** and *output*.

– **y***kmem*

The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is a waste. If this option is omitted, *sort* begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, *kmem*, *sort* will start using that number of kilobytes of memory, unless the administrative minimum or maximum is violated, in which case the corresponding extremum will be used. Thus, – **y0** is guaranteed to start with minimum memory. By convention, – **y** (with no argument) starts with maximum memory. The minimum amount is 32 Kbytes; the maximum is 32 Mbytes.

– **z***recsz*

The size of the longest line read is recorded in the sort phase so buffers can be allocated during the merge phase. If the sort phase is omitted via the – **c** or – **m** options, a popular system default size will be used. Lines longer than the buffer size will cause *sort* to terminate abnormally. Supplying the actual number of bytes in the longest line to be merged (or some larger value) will prevent abnormal termination.

The following options override the default ordering rules.

– **d** "Dictionary" order: only letters, digits and blanks (spaces and tabs) are significant in comparisons.

– **f** Fold lower case letters into upper case.

– **i** Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.

– **M** Compare as months. The first three non-blank characters of the field are folded to upper case and compared so that "JAN" < "FEB" < ... < "DEC". Invalid fields compare low to "JAN". The – **M** option implies the – **b** option (see below).

– **n** An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The – **n** option implies the – **b** option (see below). Note that the – **b** option is only effective when restricted sort key specifications are in effect.

– r Reverse the sense of comparisons.

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

The notation *+pos1 – pos2* restricts a sort key to one beginning at *pos1* and ending at *pos2*. The characters at positions *pos1* and *pos2* are included in the sort key (provided that *pos2* does not precede *pos1*). A missing *– pos2* means the end of the line.

Specifying *pos1* and *pos2* involves the notion of a field, a minimal sequence of characters followed by a field separator or a new-line. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field. The treatment of field separators can be altered using the options:

– t*x* Use *x* as the field separator character; *x* is not considered to be part of a field (although it may be included in a sort key). Each occurrence of *x* is significant (e.g., *xx* delimits an empty field).

– b Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the **– b** option is specified before the first *+pos1* argument, it will be applied to all *+pos1* arguments. Otherwise, the **b** flag may be attached independently to each *+pos1* or *– pos2* argument (see below).

Pos1 and *pos2* each have the form *m.n* optionally followed by one or more of the flags **bdfinr**. A starting position specified by *+m.n* is interpreted to mean the *n*+ 1st character in the *m*+ 1st field. A missing *.n* means .0, indicating the first character of the *m*+ 1st field. If the **b** flag is in effect *n* is counted from the first non-blank in the *m*+ 1st field; *+m.***0b** refers to the first non-blank character in the *m*+ 1st field.

A last position specified by *– m.n* is interpreted to mean the *n*th character (including separators) after the last character of the *m th* field. A missing *.n* means .0, indicating the last character of the *m*th field. If the **b** flag is in effect *n* is counted from the last leading blank in the *m*+ 1st field; *– m.***1b** refers to the first non-blank in the *m*+ 1st field.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

EXAMPLES

Sort the contents of *infile* with the second field as the sort key:

```
sort +1 -2 infile
```

Sort, in reverse order, the contents of *infile1* and *infile2*, placing the output in *outfile* and using the first character of the second field as the sort key:

```
sort -r -o outfile +1.0 -1.2 infile1 infile2
```

Sort, in reverse order, the contents of *infile1* and *infile2* using the first non-blank character of the second field as the sort key:

```
sort -r +1.0b -1.1b infile1 infile2
```

Print the password file (*passwd*(4)) sorted by the numeric user ID (the third colon-separated field):

```
sort -t: +2n -3 /etc/passwd
```

Print the lines of the already sorted file *infile*, suppressing all but the first occurrence of lines having the same third field (the options **– um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

sort – um + 2 – 3 infile

FILES

/usr/tmp/stm???

SEE ALSO

comm(1), join(1), uniq(1).

DIAGNOSTICS

Comments and exits with non-zero status for various trouble conditions (e.g., when input lines are too long), and for disorder discovered under the **– c** option. When the last line of an input file is missing a **new-line** character, *sort* appends one, prints a warning message, and continues.

NAME

spell, hashmake, spellin, hashcheck – find spelling errors

SYNTAX

spell [– **v**] [– **b**] [– **x**] [– **l**] [+local_file] [files]

/usr/lib/spell/hashmake

/usr/lib/spell/spellin n

/usr/lib/spell/hashcheck spelling_list

DESCRIPTION

Spell collects words from the named *files* and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no *files* are named, words are collected from the standard input.

Spell ignores most *troff*(1), *tbl*(1), and *eqn*(1) constructions.

Under the – **v** option, all words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.

Under the – **b** option, British spelling is checked. Besides preferring *centre, colour, programme, speciality, travelled,* etc., this option insists upon *-ise* in words like *standardise,* Fowler and the OED to the contrary notwithstanding.

Under the – **x** option, every plausible stem is printed with = for each word.

By default, *spell* (like *deroff*(1)) follows chains of included files (**.so** and **.nx** *troff*(1) requests), *unless* the names of such included files begin with **/usr/lib**. Under the – **l** option, *spell* 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), troff(1).

BUGS

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

The British spelling feature was done by an American.

NAME

spline – interpolate smooth curve

SYNTAX

spline [options]

DESCRIPTION

Spline takes pairs of numbers from the standard input as abscissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output (R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed., pp. 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *graph*(1).

The following *options* are recognized, each as a separate argument:

– a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

– k The constant k used in the boundary value computation:

tdefine prime2 'sup down 20 "' ndefine prime2 'sup "' y sub 0 prime2 ~=~ ky sub 1 prime2 , ~~~~ y sub n prime2 ~=~ ky sub n-1 prime2

is set by the next argument (default $k = 0$).

– n Space output points so that approximately n intervals occur between the lower and upper x limits (default $n = 100$).

– p Make output periodic, i.e., match derivatives at ends. First and last input values should normally agree.

– x Next 1 (or 2) arguments are lower (and upper) x limits. Normally, these limits are calculated from the data. Automatic abscissas start at lower limit (default 0).

SEE ALSO

graph(1).

DIAGNOSTICS

When data is not strictly monotone in x, *spline* reproduces the input without interpolating extra points.

BUGS

A limit of 1,000 input points is enforced silently.

NAME

split – split a file into pieces

SYNTAX

split [– *num*] [– **b**] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *num*-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 14 characters. If no output name is given, **x** is default.

-b indicates that *num* is not a number of lines, but a number of bytes, into which to chop the file. If the file is not ascii, use **-b** for sure.

Unless the file is exactly divisible by *num* bytes or lines, the last piece of the file will be fewer than *num* bytes or lines.

If no input file is given, or if – is given instead, the standard input file is used.

EXAMPLE

To chop a huge file named **cpio.output** into floppy-disc-size chunks:

split – 1200166 – b cpio.output

SEE ALSO

bfs(1), csplit(1).

NAME

stat – statistical network useful with graphical commands

SYNTAX

node-name [options] [files]

DESCRIPTION

Stat is a collection of command level functions (nodes) that can be interconnected using *sh*(1) to form a statistical network. The nodes reside in **/usr/bin/graf** (see *graphics*(1)). Data is passed through the network as sequences of numbers (vectors), where a number is of the form:

[sign](digits)(.digits)[e[sign]digits]

evaluated in the usual way. Brackets and parentheses surround fields. All fields are optional, but at least one of the fields surrounded by parentheses must be present. Any character input to a node that is not part of a number is taken as a delimiter.

Stat nodes are divided into four classes.

Transformers,	which map input vector elements into output vector elements;
Summarizers,	which calculate statistics of a vector;
Translators,	which convert among formats; and
Generators,	which are sources of definable vectors.

Below is a list of synopses for *stat* nodes. Most nodes accept options indicated by a leading minus (–). In general, an option is specified by a character followed by a value, such as **c5**. This is interpreted as **c** := 5 (**c** is assigned 5). The following keys are used to designate the expected type of the value:

c	characters,
i	integer,
f	floating point or integer,
file	file name, and
string	string of characters, surrounded by quotes to include a *Shell* argument delimiter.

Options without keys are flags. All nodes except *generators* accept files as input, hence it is not indicated in the synopses.

Transformers:

abs	[–**c***i*] – absolute value **c**olumns (similarly for –**c** options that follow)
af	[–**c***i* **t v**] – arithmetic function **t**itled output, **v**erbose
ceil	[–**c***i*] – round up to next integer
cusum	[–**c***i*] – cumulative sum
exp	[–**c***i*] – exponential
floor	[–**c***i*] – round down to next integer
gamma	[–**c***i*] – gamma
list	[–**c***i* **d***string*] – list vector elements **d**elimiter(s)
log	[–**c***i* **b***f*] – logarithm **b**ase

mod [−c*i* m*f*] − modulus
modulus

pair [−c*i* F*file* x*i*] − pair elements
File containing base vector, **x** group size

power [−c*i* p*f*] − raise to a power
power

root [−c*i* r*f*] − take a root
root

round [−c*i* p*i* s*i*] − round to nearest integer, .5 rounds to 1
places after decimal point, significant digits

siline [−c*i* i*f* n*i* s*f*] − generate a line given slope and intercept
intercept, number of positive integers, slope

sin [−c*i*] − sine

subset [−a*f* b*f* c*i* F*file* i*i* l*f* nl np p*f* s*i* t*i*] − generate a subset
above, below, File with master vector, interval, leave, master contains element numbers to leave, master contains element numbers to pick, pick, start, terminate

Summarizers:

bucket [−a*i* c*i* F*file* h*f* i*i* l*f* n*i*] − break into buckets
average size, File containing bucket boundaries, high, interval, low, number
Input data should be sorted

cor [−F*file*] − correlation coefficient
File containing base vector

hilo [− h l o ox oy]− find high and low values
high only, low only, option form, option form with **x** prepended, option form with **y** prepended

lreg [−F*file* i o s] − linear regression
File containing base vector, intercept only, option form for *siline*, slope only

mean [−f*f* n*i* p*f*] − (trimmed) arithmetic mean
fraction, number, percent

point [−f*f* n*i* p*f* s] − point from empirical cumulative density function
fraction, number, percent, sorted input

prod − internal product

qsort [−c*i*] − quick sort

rank − vector rank

total − sum total

var − variance

Translators:

bar [−a b f g r*i* w*i* x*f* xa y*f* ya yl*f* yh*f*] − build a bar chart
suppress axes, bold, suppress frame, suppress grid, region, width in percent, **x** origin, suppress **x**-axis label, **y** origin, suppress **y**-axis label, **y**-axis lower bound, **y**-axis high bound
Data is rounded off to integers.

hist [−**a b f g r***i* **x***f* **xa y***f* **ya yl***f* **yh***f*] − build a histogram
suppress **a**xes, **b**old, suppress **f**rame, suppress **g**rid, **r**egion, **x** origin, suppress **x**-axis label, **y** origin, suppress **y**-axis label, **y**-axis **l**ower bound, **y**-axis **h**igh bound

label [−**b c F***file* **h p r***i* **x xu y yr**] − label the axis of a GPS file
bar chart input, retain **c**ase, **l**abel **F**ile, **h**istogram input, **p**lot input, **r**otation, **x**-axis, **u**pper **x**-axis, **y**-axis, **r**ight **y**-axis

pie [−**b o p pn***i* **pp***i* **r***i* **v x***i* **y***i*] − build a pie chart
bold, values **o**utside pie, value as **p**ercentage(:=100), value as **p**ercentage(:=i), draw **p**ercent of **p**ie, **r**egion, no **v**alues, **x** origin, **y** origin
Unlike other nodes, input is lines of the form
[< **i e f c***c* >] value [label]
ignore (do not draw) slice, **e**xplode slice, **f**ill slice, **c**olor slice *c*=(**b**lack, **r**ed, **g**reen, **b**lue)

plot [−**a b c***string* **d f F***file* **g m r***i* **x***f* **xa xi***f* **xh***f* **xl***f* **xn***i* **xt y***f* **ya yi***f* **yh***f* **yl***f* **yn***i* **yt**] − plot a graph
suppress **a**xes, **b**old, plotting **c**haracters, **d**isconnected, suppress **f**rame, **F**ile containing x vector, suppress **g**rid, **m**ark points, **r**egion, **x** origin, suppress **x**-axis label, **x** interval, **x** high bound, **x** low bound, number of ticks on **x**-axis, suppress **x**-axis title, **y** origin, suppress **y**-axis label, **y** interval, **y** high bound, **y** low bound, number of ticks on **y**-axis, suppress **y**-axis title

title [−**b c l***string* **v***string* **u***string*] − title a vector or a GPS
title **b**old, retain **c**ase, **l**ower title, **u**pper title, **v**ector title

Generators:

gas [−**c***i* **i***f* **n***i* **s***f* **t***f*] − generate additive sequence
interval, **n**umber, **s**tart, **t**erminate

prime [-**c***i* **h***i* **l***i* **n***i*] − generate prime numbers
high, **l**ow, **n**umber

rand [−**c***i* **h***f* **l***f* **m***f* **n***i* **s***i*] − generate random sequence
high, **l**ow, **m**ultiplier, **n**umber, **s**eed

RESTRICTIONS

Some nodes have a limit on the size of the input vector.

SEE ALSO

graphics(1).
gps(4) in the *ROS Reference Manual.*

NAME

strings - find the printable strings in an object or other binary file

SYNTAX

strings [-] [**-o** [- *number*] file ...

DESCRIPTION

Strings looks for ASCII strings in a binary fiel. A string is any sequence of four or more printable characters ending with a newline or a null. Unless the - flag is given, *strings* only looks in the initialized data space of object files. If the **-o** flag is given, then each string is preceded by its offset in the file (in octal). If the *-number* flag is given, then number, rather than four, is used as the minimum string length.

Strings is useful for identifying random object files and many other things.

SEE ALSO

od(1)

NAME

strip – remove symbols and relocation bits

SYNTAX

strip name ...

DESCRIPTION

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

The effect of *strip* is the same as use of the **– s** option of *ld*.

FILES

/tmp/stm? temporary file

SEE ALSO

ld(1)

NAME

stty – set the options for a terminal

SYNTAX

stty [**–a**] [**–g**] [[-A] options] [[-B] options]

DESCRIPTION

Stty sets or examines the terminal I/O options for the device that is the current standard input. Without arguments, *stty* reports the settings of certain options; with the **–a** option, it reports all of the option settings; with the **–A** option, it reports all of the option settings in the AT&T System V **stty** format; with the **–B** option, it reports all of the option settings in the BSD 4.2 **stty** format; 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 below may be found in *termio*(7).

The ROS **tty** interface is the union of AT&T System V **tty** and BSD 4.2 **tty** interfaces. When setting options, you must specify which of the two interfaces is being used whenever a name conflict exists between the two interfaces. For the AT&T mode setting, the command format is:

stty [**-A**] options

For BSD, the command format is:

stty [**-B**] options

Note that the "-A" and "-B" options are to be used to force the AT&T or BSD interpretation of the options. If no option is specified, the AT&T interpretation is tried first and if no match is found, the BSD option is tried. The options listed below have been divided into three categories: the first category is the set of options common between AT&T and BSD; the second category is the set of options that apply only to the AT&T usage of terminals, and the third category is a set that only applies to the BSD 4.2 usage of terminals. Note that many combinations of options do not make sense and no sanity checking is performed.

AT&T and BSD COMMON OPTIONS

The following table lists the options that have the same meaning in both AT&T and BSD terminal interfaces. If an option can be referred to by both an AT&T and a BSD option name, then the BSD option is listed in curly brackets ({}). These options can be set using either the AT&T or BSD option name.

Control Modes

0	Hang up phone line immediately
50 75 110 134 150 200 **300 600 1200 1800 2400** **4800 9600 19200 exta extb**	Set terminal baud rate to the number given, if possible.
clocal (–clocal) **{nohang (–nohang)}**	Assume a line without (with) modem control.

Input Modes

cauto (–cauto)	Enable (disable) hardware flow control. (See termio(7)).

ixoff (–ixoff)
{tandem (–tandem)} Request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

ixany (–ixany)
{–decctlq (decctlq)} Allow any character (only ASCII DC 1) to restart output.

iuclc (–iuclc)
{lcase (–lcase)} Map (do not map) upper-case alphabetics to lower case on input.

Output Modes

cr0 cr1 cr2 cr3 Select style of delay for carriage returns (see termio(7)).

nl0 nl1 Select style of delay for linefeeds (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)).

opost (–opost)
{–litout (litout)} Post-process output (do not post-process output; ignore all other output modes).

vt0 vt1 Select style of delay for vertical tabs (see termio(7)).

Local Modes

echo (–echo) Echo back (do not echo back) every character typed.

echoe (–echoe)
{crterase (–crterase)} Echo (do not echo) ERASE character as a backspace-sp-backspace string.

echoi (–echoi) Echo characters when typed (read).

echok (–echok)
{crtkill (–crtkill)} Echo NL after kill character.

noflsh (–noflsh) Disable (enable) flush after INTR or QUIT.

lintrup (–lintrup)
{intrup (–intrup)} Send (do not send) signal SIGIO when input is available. (See **termio**(7)).

Control Assignments

erase *c* Set erase character to *c*.

kill *c* Set kill character to *c*.

intr *c* — Set interrupt character to *c*.

quit *c* — Set quit character to *c*.

eof *c* — Set end of file character to *c*.

eol *c* — Set end of line character to *c*.

{brk *c*} — Set an extra new line character to *c*.

ek — Reset ERASE and KILL characters back to normal (control-h and control-x).

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 a DEL and "^-" is undefined.

AT&T SYSTEM V ONLY OPTIONS

The second set of options are used to enable or disable AT&T features of the terminal interface. They do not have a corresponding feature or option in the BSD terminal interface with the same semantics.

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)). **hupcl (–hupcl)** Hang up (do not hang up) a modem 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.

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.
ixon (–ixon)* Enable (disable) START/STOP output control. Output is stopped by sending an ASCII DC3 and started by sending an ASCII DC1.

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.
ocrcrnl (–ocrcrnl)* Convert (do not convert) CR to CR-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) NOT CURRENTLY IMPLEMENTED.
ofdel (–ofdel) NOT CURRENTLY IMPLEMENTED.

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.
echonl (**—echonl**)	NOT CURRENTLY IMPLEMENTED.

Control Assignments

control-character c	Set *control-character* to *c*, where *control-character* is **min**, or **time** (**min** and **time** are used with **—icanon**; see *termio*(7)). If *c* is preceded by an (escaped from the shell) caret (^), then the value used is the corresponding CTRL character (e.g., "^**d**" is a **CTRL-d**); "^**?**" is a DEL and "^—" is undefined.

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). Please note that this option will set (unset) the BSD "raw" flag unless used with the "-A" prefix.
nl (**—nl**)	Unset (set) **icrnl**, **onlcr**. In addition **—nl** unsets **inlcr**, **igncr**, **ocrnl**, and **onlret**. Please note that this option will unset (set) the BSD CRHOD flag unless used with the "-A" option.
lcase (**—lcase**)	Set (unset) **xcase**, **iuclc**, and **olcuc**. Please note that this option is not the same as the BSD "lcase" option.
LCASE (**—LCASE**)	Same as **lcase** (**—lcase**).
tabs (**—tabs** or **tab3**)	Preserve (expand to spaces) tabs when printing.
sane	Resets all modes to some reasonable values (parenb¦cs7¦cread¦brkint¦ignpar¦istrip¦ icrnl¦ixon¦isig¦icanon¦echo¦echoe¦opost¦onlcr¦tab3 and all characters to the default values).
term	Set all modes suitable for the terminal type *term*, where *term* is one of **tty33**, **tty37**, **vt05**, **tn300**, **ti700**, or **tek**.

* These options apply only to processes executing in the AT&T System V domain.

BSD 4.2 ONLY OPTIONS

The third set of options are used to enable or disable BSD features of the terminal interface. They do not have a corresponding feature or option in the AT&T terminal interface with the same semantics.

all	Display all BSD normally used option settings.
everything	Display all BSD options that stty knows about.
even (**—even**)	Allow (disallow) even parity input.
odd (**—odd**)	Allow (disallow) odd parity input.
raw (**—raw**)**	Set (unset) raw mode input (no input processing (erase, kill, interrupt)).

cooked**	Same as "-raw"
cbreak (—cbreak)**	Make each character available to *read*(2) as received; no erase and kill processing but all other processing (interrupt, suspend, ...) is performed. (make characters available to *read* only when newline is received).
—nl (nl)**	Allow carriage return for new line, and output CR-NL for carriage return or new line (accept only new line to end lines).
—tabs (tabs)	Replaces tabs by spaces when printing (preserve tabs).
sane	Resets all modes to some reasonable values (new¦crt¦-nl¦echo¦even/odd¦nofish and all characters to defaults values).
new	Use new driver (enable job control processing).
crt	Set options for a CRT (crtbs, ctlecho and if >= 1200 baud , crterase, and crtkill.)
crtbs	Echo backspaces on erase characters.
prterase	For printing terminals, echo erase characters backwards within "\" and "/".
ctlecho (—ctlecho)	Echo control characters as ^*x* and delete as ^*?* (as themselves otherwise).
tostop (—tostop)	Background jobs stop if they attempt terminal output (otherwise allowed to produce output).
flusho (-flusho)	Output is (not) being discarded, usually because the user pressed control-o.
pendin (—pendin)	Input is (not) pending after a switch from cbreak to cooked and will be re-input when a read becomes pending or more input arrives.
tilde (—tilde)	Convert "~" to "`" on output (or leave it alone).
stop *c*	Set stop character to *c*.
start *c*	Set start character to *c*.
susp *c*	Set suspend process character to *c* (default control-z).
dsusp *c*	Set delayed suspend process character to *c* (default control-y).
rprnt *c*	Set reprint line character to *c* (default control-r).
flush *c*	Set flush output character to *c* (default control-o).
werase *c*	Set word erase character to *c* (default control-w).
lnext *c*	Set literal next character to *c* (default control-v).

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 a DEL and "^-" is undefined.

** These options apply only to processes executing in the BSD 4.2 domain.

EXAMPLES

To set the baud rate on tty3 to 19,200, enter: **stty 19200 < /dev/tty3**.
To set the "echo erase on backspace" on tty3 enter either:
stty echoe < /dev/tty3 or
stty crterase < /dev/tty3.
To set the BSD 4.2 "raw" flag and the characters echoing as immediate, and the (AT&T) number of stop bits to 2:
stty -B raw echoi -A stopb, or
stty stopb -B raw echoi.
Note that "-A" and "-B" are used to force the AT&T or BSD interpretation of the options, otherwise AT&T is tried first and if no match is found, the BSD option is tried.
The command **stty sane** is equivalent to **stty -A sane -B sane**.

LIMITATIONS

The **−g** option works only for the AT&T termio structure.

SEE ALSO

tabs(1), ioctl(2), termio(7), tty(7)

NAME

style – analyze writing style

SYNTAX

style [– **ml**] [– **m***x*] [– **a**] [– **e**] [– **l** num] [– **r** num]
[– **p**] [– **P**] file ...

DESCRIPTION

Style analyzes an unformatted text for readability, sentence length and structure, word length and usage, verb type, and sentence openers.

Before analyzing the text, *style* removes all formatting commands by means of *deroff(1)*.

– **ml** causes *deroff(1)* to skip lists; use this if the document contains many lists of non-sentences.

– **m***x* is a macro package like **-mm, -ms, -man**, or **-me**. The input is **not** formatted according to the macro package, but macro flags should be included because the packages might contain text that should be examined.

– **a** print all sentences with their length and readability index.

– **e** print all sentences that begin with an expletive.

– **p** print all sentences that contain a passive verb.

– **l***num* print all sentences longer than *num*.

– **r***num* print all sentences whose readability index is greater than *num*.

– **P** print parts of speech of the words in the document.

SEE ALSO

deroff(1), diction(1), soelim(1)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks.

NAME

su – become super-user or another user

SYNTAX

su [–] [name [arg ...]]

DESCRIPTION

Su allows you 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** 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(1), sh(1), environ(5).

NOTES

When one does an *su*, the current directory is not searched. This protects the su'ed user from randomly executing programs in user directories that may have the same name as common system utilities.

NAME

sum – print checksum and block count of a file

SYNTAX

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.

Block counts are calculated by: Int(((file size(bytes)+ 4095)/4096)*4. The block count equation evaluates to a 1024-byte increment.

SEE ALSO

wc(1).

DIAGNOSTICS

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

NAME

symorder – rearrange name list

SYNTAX

symorder orderlist symbolfile

DESCRIPTION

Orderlist is a file containing symbols to be found in symbolfile, 1 symbol per line.

Symbolfile is updated in place to put the requested symbols first in the symbol table, in the order specified. This is done by swapping the old symbols in the required spots with the new ones. If all of the order symbols are not found, an error is generated.

SEE ALSO

nlist(3)

NAME

tabs – set tabs on a terminal

SYNTAX

tabs [tabspec] [**+mn**] [**– 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., "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 "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

SYNTAX

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

SYNTAX

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 on stderr 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 on stderr the name of each file it treats, preceded by the function letter. With the **t** function, **v** gives more information about the tape entries: protection bits, user and group id, file size in bytes, last modification date, and file 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/rmt?**. 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* to not restore the modification times. The modification time of the file will be the time of extraction.

FILES

/dev/rmt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if enough memory is not available to hold the link tables.

BUGS

There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow.
The **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

tee – pipe fitting

SYNTAX

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

telnet – user interface to the TELNET protocol

SYNTAX

telnet [host [port]]

DESCRIPTION

Telnet is used to communicate with another host using the TELNET protocol. If *telnet* is invoked without arguments, it enters command mode, indicated by its prompt (telnet>). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an *open* command (see below) with those arguments.

Once a connection has been opened, *telnet* enters input mode. In this mode, any text typed is sent to the remote host. To issue *telnet* commands when in input mode, precede them with the *telnet* escape character (initially ^[). When in command mode, the normal terminal editing conventions are available.

The following commands are available. Only enough of each command to uniquely identify it need be typed.

open *host* [*port*]
: Open a connection to the named host. If the no port number is specified, *telnet* will attempt to contact a TELNET server at the default port. The host specification may be either a host name (see *hosts*(4)) or an Internet address specified in the dot notation.

close Close a TELNET session and return to command mode.

quit Close any open TELNET session and exit *telnet*.

z Suspend *telnet*. This command only works when the user is using the *csh*(1).

escape [*escape-char*]
: Set the *telnet* escape character. Control characters may be specified as ^ followed by a single letter; e.g. control-X is ^X.

status Show the current status of *telnet*. This includes the peer one is connected to, as well as the state of debugging.

options
: Toggle on and off the display of the TELNET options being processed. When option viewing is enabled, all TELNET option negotiations will be displayed. Options sent by *telnet* are displayed as SENT, while options received from the TELNET server are displayed as RCVD.

crmod Toggle return mode. When this mode is enabled, any return characters received from the remote host will be mapped into a return and a line feed. This mode does not affect those characters typed by the user, only those received. This mode is not very useful, but is required for some hosts that ask the user to do local echoing.

? [*command*]
: Get help. With no arguments, *telnet* prints a help summary. If a command is specified, *telnet* will print the help information available about the command only.

BUGS

This implementation is very simple because *rlogin*(1) is the standard mechanism used to communicate locally with hosts.

NAME

telnetd – DARPA TELNET protocol server

SYNTAX

/etc/telnetd [**–d**] [*port*]

DESCRIPTION

Telnetd is a server which supports the DARPA standard TELNET virtual terminal protocol. The TELNET server operates at the port indicated in the "telnet" service description; see *services*(4). This port number may be overridden (for debugging purposes) by specifying a port number on the command line. If the **–d** option is specified, each socket created by *telnetd* will have debugging enabled.

Telnetd operates by allocating a pseudo-terminal device (see *pty*(7)) for a client, then creating a login process which has the slave side of the pseudo-terminal as **stdin**, **stdout**, and **stderr**. *Telnetd* manipulates the master side of the pseudo terminal, implementing the TELNET protocol and passing characters between the client and login process.

When a TELNET session is started up, *telnetd* sends a TELNET option to the client side, indicating a willingness to do "remote echo" of characters. The pseudo terminal allocated to the client is configured to operate in "cooked" mode with XTABS and CRMOD enabled (see *termio*(7)). Aside from this initial setup, the only mode changes *telnetd* will carry out are those required for echoing characters at the client side of the connection.

Telnetd supports binary mode, as well as most of the common TELNET options, but does not, for instance, support timing marks.

SEE ALSO

telnet(1)

NAME

test – condition evaluation command

SYNTAX

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.

NAME

tic – terminfo compiler

SYNTAX

tic [– **v**[*n*]] [– **t** *termtype*] *file* ... [– **d** *destination*]

DESCRIPTION

Tic translates terminfo files from the source format into the compiled format. The results are placed in the directory **/usr/lib/terminfo**.

– **v** *n* sets the diagnostic verbosity level to *n*. *n* is optional.

– **t** *termtype*

Tic searches for the *termtype* entry in the source *file* and compiles only that entry (and associated names). EXAMPLE: if *termtype* is **hp2645**, and there is an entry **hp2645 | 2645 |hp4s**, tic makes an entry for all three names.

– **d** *destination*

re-routes the compiled result from **/usr/lib/terminfo** to the *destination* directory, and creates *destination* if necessary.

Tic compiles all terminfo descriptions in the given files. When a **use=** field is discovered, **tic** searches first the current file, then the master file, which is "./terminfo.src".

If the environment variable TERMINFO is set, the results are placed in the directory named by TERMINFO, instead of in **/usr/lib/terminfo**. The directory named by TERMINFO must already exist.

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/*/* compiled terminal capability data base

SEE ALSO

curses(3X), terminfo(4).

BUGS

Instead of searching **./terminfo.src**, it should check for an existing compiled entry.

NAME

time – time a command

SYNTAX

time command

DESCRIPTION

Time executes *command* and then prints the elapsed time during the command, and the command execution time.

Times are reported in the form *hours:minutes:seconds:tenths-of-second* on the standard error output.

SEE ALSO

times(2).

NAME

toc – graphical table of contents routines

SYNTAX

dtoc [directory]
ttoc mm-file
vtoc [**–cdhnimsvn**] [TTOC file]

DESCRIPTION

All of the commands listed below reside in **/usr/bin/graf** (see *graphics*(1G)).

dtoc Dtoc makes a textual table of contents, TTOC, of all subdirectories beginning at *directory* (*directory* defaults to .). The list has one entry per directory. The entry fields from left to right are level number, directory name, and the number of ordinary readable files in the directory. *Dtoc* is useful in making a visual display of all or parts of a file system. The following will make a visual display of all the readable directories under **/**:

dtoc / ¦ vtoc ¦ td

ttoc Output is the table of contents generated by the .TC macro of *mm*(1) translated to TTOC format. The input is assumed to be an *mm* file that uses the .H family of macros for section headers. If no *file* is given, the standard input is assumed.

vtoc *Vtoc* produces a GPS describing a hierarchy chart from a TTOC. The output drawing consists of boxes containing text connected in a tree structure. If no *file* is given, the standard input is assumed. Each TTOC entry describes one box and has the form:

id [*line-weight,line-style*] "*text*" [*mark*]

where:

id is an alternating sequence of numbers and dots. The *id* specifies the position of the entry in the hierarchy. The *id* **0.** is the root of the tree.

line-weight is either:

n, normal-weight; or
m, medium-weight; or
b, bold-weight.

line-style is either:

so, solid-line;
do, dotted-line;
dd, dot-dash line;
da, dashed-line; or
ld, long-dashed

text is a character string surrounded by quotes. The characters between the quotes become the contents of the box. To include a quote within a box, it must be escaped (**\"**).

mark is a character string (surrounded by quotes if it contains spaces), with included dots being escaped. The string is put above the top right corner of the box. To include either a quote or a dot within a *mark*, it must be escaped.

Entry example: 1.1 b,da "ABC" DEF
Entries may span more than one line by escaping the new-line (**\new-line**).

Comments are surrounded by the **/*,*/** pair. They may appear anywhere in a

TTOC.

Options:

c Use text as entered (default is all upper case).

d Connect the boxes with diagonal lines.

h*n* Horizontal interbox space is *n*% of box width.

i Suppress the box *id*.

m Suppress the box *mark*.

s Do not compact boxes horizontally.

v*n* Vertical interbox space is *n*% of box height.

SEE ALSO

graphics(1G).
gps(4) in the *ROS Reference Manual*.

NAME

touch – update access and modification times of a file

SYNTAX

touch [**–amc**] [[yy]mmddhhmm] 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

tplot – graphics filters

SYNTAX

tplot [– **T***terminal* [– **e** raster]]

DESCRIPTION

These commands read plotting instructions (see *plot*(4)) from the standard input and in general produce, on the standard output, plotting instructions suitable for a particular *terminal*. If no *terminal* is specified, the environment parameter $TERM (see *environ*(5)) is used. Known *terminal*s are:

300	DASI 300.
300S	DASI 300s.
450	DASI 450.
4014	Tektronix 4014.
r15	Ridge Monochrome Display
ver	Versatec V80. This version of *plot* places a scan-converted image in **/usr/tmp/raster$$** and sends the result directly to the plotter device, rather than to the standard output. The – **e** option causes a previously scan-converted file *raster* to be sent to the plotter.

FILES

/usr/lib/t300
/usr/lib/t300s
/usr/lib/t450
/usr/lib/t4014
/usr/lib/vplot
/usr/lib/r15
/usr/tmp/raster$$

SEE ALSO

plot(3X), plot(4), term(5).

NAME

tput – query terminfo database

SYNTAX

tput [**-T**type] capname

DESCRIPTION

Tput uses the *terminfo(4)* database to make terminal-dependent capabilities and information available to the shell. *Tput* outputs a string if the attribute (**capability name**) is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, tput simply sets the exit code (0 for TRUE, 1 for FALSE), and does no output.

-T*type* indicates the type of terminal. Normally this flag is unnecessary, as the default is taken from the environment variable **$TERM.**

Capname indicates the attribute from the *terminfo* database. See *terminfo(4)*.

EXAMPLES

tput clear	Echo clear-screen sequence for the current terminal.
tput cols	Print the number of columns for the current terminal.
tput -T450 cols	Print the number of columns for the 450 terminal.
bold=`tput smso`	Set shell variable "bold" to stand-out mode sequence for current terminal. This might be followed by a prompt: **echo "${bold}Please type in your name: \c"**
tput hc	Set exit code to indicate if current terminal is a hardcopy terminal.

FILES

/etc/term/?/*	Terminal descriptor files
/usr/include/term.h	Definition files
/usr/include/curses.h	

DIAGNOSTICS

Tput prints error messages and returns the following error codes on error:

-1	Usage error.
-2	Bad terminal type.
-3	Bad capname.

In addition, if a capname is requested for a terminal that has no value for that capname (e.g., **tput -T450 lines**), **-1** is printed.

SEE ALSO

stty(1), terminfo(4).

NAME

tr – translate characters

SYNTAX

tr [– **cds**] [string1 [string2]]

DESCRIPTION

Tr deletes specified input characters, or does an in-order mapping of the characters in *string1* to the characters in *string2*.

– **c** Complement. *string1* will be equivalent to every ASCII character, in the ASCII code range 1 to 255 (1 to 377 octal) **except** the ones specified.

– **d** Deletes all input characters in *string1*. IGNORES *string2*.

– **s** Squeezes repititions of output characters that are in *string2* to single characters and IGNORES *string1*, or repetitions of characters found in *string1* if there is no *string2*.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[**a– z**] Represents a range of characters as ordered in the ASCII table.

[**a***n*] Represents *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* represents an indefinite number. When the first string specifies a range of characters, and you don't know how many it would match but want to replace all of them with a certain other character, use * or *0 in string2 to represent "as many as necessary."

Use \ to remove any special meaning from a character in a string. To represent ASCII characters by their octal codes, use \ followed by 1, 2, or 3 octal digits for the character.

EXAMPLES

To change all b's to k's: **tr b k** < *myfile*

To change all b's to k's and all p's to e's: **tr bp ke** < *myfile*

To get rid of all b's and squeeze the f's: **tr -ds b f**

To print only the punctuation from a text: **tr -d "[A-Z][a-z]"** < *text*

The strings are quoted to protect the special characters from interpretation by the shell.

To create a list of alphabetic-only words (one per line) derived from *text*:

tr – cs "[A– Z][a– z]" "[\012*]" < text

(012 is the octal ASCII code for newline.) In English: Translate everything but alphabetics into as many newline characters as may be required, but squeeze (-s) the excessive repetitions of newline characters into just one newline.

Leave out the -s and excessive newlines appear because they are not squeezed. Leave out the -c and alphabetic strings (words) are turned into newlines, and only the punctuation will be output.

SEE ALSO

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

BUGS

Won't handle ASCII NUL in either string, and always deletes NUL from input.

NAME

true, false – provide truth values

SYNTAX

true

false

DESCRIPTION

True does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1) such as:

```
while true
do
        command
done
```

SEE ALSO

sh(1).

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

tsort – topological sort

SYNTAX

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.

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

BUGS

Uses a quadratic algorithm.

NAME

tty – get the terminal's name

SYNTAX

tty [**– s**]

DESCRIPTION

Tty prints the path name of the user's terminal. 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

Tty prints "not a tty" if the standard input is not a terminal and **– s** is not specified.

NAME

ul – do underlining

SYNTAX

ul [– **i**] [– **t** *terminal*] [*name* ...]

DESCRIPTION

Ul reads the named files (or standard input if none are given) and translates occurrences of underscores to the sequence which indicates underlining for the terminal in use, as specified by the environment variable TERM. The – **t** option overrides the terminal kind specified in the environment.

Ul uses curses(3X) and terminfo(4) to determine the appropriate sequences for underlining. If the terminal is incapable of underlining, but is capable of a standout mode then that is used instead. If the terminal can overstrike, or handles underlining automatically, *ul* degenerates to *cat*(1). If the terminal cannot underline, underlining is ignored.

The – **i** option causes *ul* to indicate underlining onto by a separate line containing appropriate dashes '– '; this is useful when you want to look at the underlining which is present in an *nroff* output stream on a crt-terminal.

SEE ALSO

man(1), nroff(1), colcrt(1)

BUGS

Nroff usually outputs a series of backspaces and underlines intermixed with the text to indicate underlining. No attempt is made to optimize the backward motion.

NAME

umask – set file-creation mode mask

SYNTAX

umask [ooo]

DESCRIPTION

The user file-creation mode mask is set to *ooo*. The three octal digits refer to read/write/execute permissions for *owner*, *group*, and *others*, respectively (see *chmod*(2) and *umask*(2)). The value of each specified digit is subtracted from the corresponding "digit" specified by the system for the creation of a file (see *creat*(2)). For example, **umask 022** removes *group* and *others* write permission (files normally created with mode **777** become mode **755**; files created with mode **666** become mode **644**).

If *ooo* is omitted, the current value of the mask is printed.

Umask is recognized and executed by the shell.

SEE ALSO

chmod(1), sh(1), chmod(2), creat(2), umask(2).

NAME

uname – print name of current UNIX System

SYNTAX

uname [– **snrvma**]

DESCRIPTION

Uname prints the current system name of the UNIX System on the standard output file. It is mainly useful to determine what system one is using. The options cause selected information returned by *uname*(2) to be printed:

– **s** print the system name (default).

– **n** print the nodename (the nodename may be a name that the system is known by to a communications network).

– **r** print the operating system release.

– **v** print the operating system version.

– **m** print the machine hardware name.

– **a** print all the above information.

Arguments not recognized default the command to the – **s** option.

SEE ALSO

uname(2).

NAME

unget – undo a previous get of an SCCS file

SYNTAX

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

SYNTAX

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

SYNTAX

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

uptime – show how long system has been up

SYNTAX

uptime

DESCRIPTION

Uptime prints the current time, the length of time the system has been up, the number of users logged into the system, and the load averages. The load average numbers give the number of processes on the list of processes ready to run (ready list), plus the number of processes that are currently faulted (due to a page fault, for instance). Load average numbers are averaged over 1, 5, and 15 minutes.

FILES

/usr/bin/uptime

NAME

users – compact list of users who are on the system

SYNTAX

users

DESCRIPTION

Users lists the login names of the users currently on the system in a compact, one-line format.

FILES

/etc/utmp

SEE ALSO

who(1)

NAME

uuclean – uucp spool directory cleanup

SYNTAX

/usr/lib/uucp/uuclean [*options*]

DESCRIPTION

Uuclean deletes uucp files according to their names or ages in hours.

– **d***directory*
clean *directory* instead of the spool directory.

– **p***pre* delete any files with the *pre* prefix. Up to 10 – **p** arguments may be specified. – **p** with no arguments causes all files older than than *time* hours to be removed.

– **n***time*
only if – **p** is specified first, files whose age is more than *time* hours are removed. – **n** with no argument sets *time* to 72 hours.

– **m***file* specifies that removed mail items are sent to a file named *file*. – **m** with no argument specifies that mail is sent to *file*.

Uuclean(1) is typically invoked by **cron(1)**.

FILES

/usr/lib/uucp directory with internally-used uuclean commands
/usr/spool/uucp spool directory

SEE ALSO

cron(1), uucp(1), uux(1)

NAME

uucp – unix to unix copy

SYNTAX

uucp [option] ... source-file ... destination-file

DESCRIPTION

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!pathname

where 'system-name' is taken from a list of system names which *uucp* knows about. Shell metacharacters ? *[] appearing in the pathname part will be expanded on the appropriate system.

Pathnames may be one of

(1) a full pathname;

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

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

If the result is an erroneous pathname for the remote system the copy will fail. If the destination-file is a directory, the last part of the source-file name is used.

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

The following options are interpreted by *uucp*.

– **d** Make all necessary directories for the file copy.

– **c** Use the source file when copying out rather than copying the file to the spool directory.

– **m** Send mail to the requester when the copy is complete.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/* - other data and program files

SEE ALSO

uux(1), mail(1)
Uucp description in the *ROS Utility Guide*

NOTE

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 pathname; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary pathnames.

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

uuencode,uudecode – encode/decode a binary file for transmission via mail

SYNTAX

uuencode [source] remotedest |**mail** sys1!sys2!..!decode
uudecode [file]

DESCRIPTION

Uuencode and *uudecode* are used to send a binary file via uucp (or other) mail. This combination can be used over indirect mail links even when *uusend*(1) is not available.

Uuencode takes the named source file (default standard input) and produces an encoded version on the standard output. The encoding uses only printing ASCII characters, and includes the mode of the file and the *remotedest* for recreation on the remote system.

Uudecode reads an encoded file, strips off any leading and trailing lines added by mailers, and recreates the original file with the specified mode and name.

The encode file has an ordinary text form and can be edited by any text editor to change the mode or remote name.

SEE ALSO

uuencode(4), uusend(1), uucp(1), uux(1), mail(1)

BUGS

The file is expanded by 35% (3 bytes become 4 plus control information) causing it to take longer to transmit.

The user on the remote system who is invoking *uudecode* (often *uucp)* must have write permission on the specified file.

NAME

uulog – log UUCP transactions

SYNTAX

uulog [option] ...

DESCRIPTION

Uulog maintains a summary log of *uucp* and *uux*(1) transactions in the file '/usr/spool/uucp/LOGFILE' by gathering information from partial log files named '/usr/spool/uucp/LOG.*.?'. It removes the partial log files.

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*.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/* - other data and program files

SEE ALSO

Uucp description in the *ROS Utility Guide*

NAME

uupoll – poll a system for UUCP work to do

SYNTAX

uupoll *system*

DESCRIPTION

Uupoll attempts to establish connection with the named system, even if recent attempts have failed. *Upoll* is a shortcut to invoking the *uucico(1)* command directly.

FILES

/usr/lib/uucp/L.sys

SEE ALSO

uucp(1), uux(1)

NAME

uusend – send a file to a remote host

SYNTAX

uusend [– **m** mode] sourcefile sys1!sys2!..!remotefile

DESCRIPTION

Uusend sends a file to a given location on a remote system. The system need not be directly connected to the local system, but a chain of *uucp*(1) links needs to connect the two systems.

If the – **m** option is specified, the mode of the file on the remote end will be taken from the octal number given. Otherwise, the mode of the input file will be used.

The sourcefile can be "– ", meaning to use the standard input. Both of these options are primarily intended for internal use of uusend.

The remotefile can include the ~userid syntax defined in *uucp(1)*.

DIAGNOSTICS

If anything goes wrong any further away than the first system down the line, you will never hear about it.

SEE ALSO

uux(1), uucp(1), uuencode(1)

BUGS

This command shouldn't exist, since *uucp* should handle it.

All systems along the line must have the *uusend* command available and allow remote execution of it.

Some uucp systems have a bug where binary files cannot be the input to a uux command. If this bug exists in any system along the line, the file will show up severly munged.

NAME

uusnap – show snapshot of the UUCP system

SYNTAX

uusnap

DESCRIPTION

Uusnap displays in tabular format a synopsis of the current UUCP situation. The format of each line is as follows:

site N Cmds N Data N Xqts Message

Where "site" is the name of the site with work, "N" is a count of each of the three possible types of work (command, data, or remote execute), and "Message" is the current status message for that site as found in the STST file.

Included in "Message" may be the time left before UUCP can re-try the call, and the count of the number of times that UUCP has tried to reach the site.

SEE ALSO

uucp(1), *UUCP* description in the *ROS Utility Guide*

NAME

uux – unix to unix command execution

SYNTAX

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(1), uucp(1), and the *UUCP* description in the *ROS Utility Guide.*

BUGS

The use of the shell metacharacter * will probably not do what you want it to do. The shell tokens << and >> are not implemented.

Only the first command of a shell pipeline may have a *system-name*. All other commadns are executed on the system of the first command.

NAME

val – validate SCCS file

SYNTAX

val –

val [– **s**] [– **r**SID] [– **m**name] [– **y**type] files

DESCRIPTION

Val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of keyletter arguments, which begin with a – , and named files.

Val has a special argument, – , which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

Val generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

The keyletter arguments are defined as follows. The effects of any keyletter argument apply independently to each named file on the command line.

– **s** The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

– **r***SID* The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., **r1** is ambiguous because it physically does not exist but implies **1.1**, **1.2**, etc. which may exist) or invalid (e. g., **r1.0** or **r1.1.0** 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

If *Val* attempts to process more than 50 files on a single command line, it fails and the debugger will be activated.

NAME

vc – version control

SYNTAX

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 – screen-oriented (visual) display editor based on ex

SYNTAX

vi [– **t** *tag*] [– **r** *file*] [– **l**] [– **wn**] [– **x**] [– **R**]
[+ *command*] name ...

view [**-t** *tag*] [– **r** *file*] [– **l**] [– **wn**] [– **x**] [– **R**]
[+ *command*] name ...

DESCRIPTION

Vi (visual) is a full screen-oriented text editor. Ex(1) editing commands are a subset of *vi* and are available for use within *vi*.

With **vi**, changes to the file are reflected on the terminal screen. The cursor on the screen indicates the position in the file where editing commands will be executed.

Refer to the material on **Ex** and **Vi** in the *ROS Text Editing Guide* for full details on the use of **vi**.

INVOCATION

The following invocation options are interpreted by *vi*:

– **t** *tag*	Edit the file containing the *tag* and position the cursor at its location.
– **r***file*	Recover *file* after an editor or system crash. If *file* is not specified, a list of all saved files will be printed.
– **l**	LISP mode; indents appropriately for lisp code, the () {} [[and]] commands in *vi* and *open* are modified to have meaning for *lisp* .
– **w***n*	Set the default window size to *n*. This is useful when using the editor over a slow speed line.
– **x**	Encryption mode; vi prompts for a keyword to allow creating of editing an encrypted file. See crypt(1).
– **R**	Read-only mode; to protect from accidental overwriting of the file, a modified version of the file cannot be saved.
+ *command*	The specified *ex* command is interpreted before editing begins.

The *name* argument indicates files to be edited.

View is the same as **vi -R**; a modified version of the file cannot be saved.

VI MODES

Command	Normal and initial mode. Other modes return to command mode upon completion. Press ESC to cancel a command that has been partially entered.
Input	Entered by **a i A I o O c C s S R**. Arbitrary text may then be entered. Input mode is normally terminated with ESC character, or abnormally with interrupt.
Last line	Reading input for **:** **/** **?** or **!**; terminate with CR to execute, interrupt to cancel.

COMMAND SUMMARY

Sample commands

← ↓ ↑ →	arrow keys move the cursor
h j k l	same as "left","down","up",and "right" arrows
i*text*ESC	insert text *abc*
cw*new*ESC	change word to **new**

ea*s*ESC	move to end of word, add an **s**
x	delete a character
dw	delete a word
dd	delete a line
3dd	delete 3 lines
u	undo previous change
ZZ	exit the editor, saving changes
:q![RET]	quit the editor, discarding changes
*/***text**CR	search for *text*
^U ^D	scroll up or down
:*ex cmd***CR**	any ex or ed command

Counts before vi commands

Numbers may be typed as a prefix to some commands. They are interpreted in one of these ways.

line/column number	**z G |**
scroll amount	**^D ^U**
repeat effect	most of the rest

Interrupting, Canceling

ESC	end an i, a, A, s, S, or R action; or cancel a command
^?	(delete or rubout) interrupts
^L	reprint screen if **^?** scrambles it
^R	reprint screen if ^L is → key

File Manipulation (press [RETURN] after each)

:w	write (save) cumulative changes to file
:q	quit
:q!	quit, discard changes
:e *name*	edit file *name*
:e!	reedit, discard changes
:e + *name*	edit, starting at end
:e +*n*	edit starting at line *n*
:e #	edit alternate file
	synonym for **:e #**
:w *name*	write this file and call it *name*
:w! *name*	overwrite this file on file *name*
:sh	run shell, then return to editor
:!*cmd*	run *cmd*, then return to editor
:n	edit next file in arglist
:n *args*	specify new arglist
^G	show current file name and cursor line number
:f	show current file name and cursor line number
:ta *tag*	to tag file entry *tag*
^]	**:ta**, following word is *tag*

In general, any ex(1) or ed(1) command may be entered in the form: :*command* [RET].

Positioning within file

^F	forward screen
^B	backward screen
^D	scroll down half screen
^U	scroll up half screen
G	go to specified line (end default)
/*pat*	next line matching *pat*
?*pat*	prev line matching *pat*
n	repeat last / or ?
N	reverse last / or ?
/*pat*/+*n*	noth line after *pat*
?*pat*?−*n*	noth line before *pat*
]]	next section/function
[[	previous section/function
(	beginning of sentence
)	end of sentence
{	beginning of paragraph
}	end of paragraph
%	find matching ( ) { or }

Adjusting the screen

^L	clear and redraw
^R	retype, eliminate @ lines
z[RET]	redraw, current at window top
z− [RET]	... at bottom
z.[RET]	... at center
/*pat*/z− [RET]	*pat* line at bottom
z*n*.[RET]	use *n* line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and returning

``	move cursor to previous context
''	... at first non-white in line
m*x*	mark current position with letter *x*
`*x*	move cursor to mark *x*
'*x*	... at first non-white in line

Line positioning

H	top line on screen
L	last line on screen
M	middle line on screen
+	next line, at first non-white
−	previous line, at first non-white
[RET]	return, same as +
↓ or j	next line, same column
↑ or k	previous line, same column

Character positioning

^	first non white
0	beginning of line
$	end of line
h or →	forward
l or ←	backwards
^H	same as ←
space	same as →
f*x*	find *x* forward
F*x*	**f** backward
t*x*	upto *x* forward
T*x*	back upto *x*
;	repeat last **f F t** or **T**
,	inverse of **;**
|	to specified column
%	find matching ({) or }

Words, sentences, paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back **W**
E	to end of **W**

Commands for LISP Mode

)	Forward s-expression
}	... but do not stop at atoms
(	Back s-expression
{	... but do not stop at atoms

Corrections during insert

^H	erase last character
^W	erase last word
erase	your erase, same as **^H**
kill	your kill, erase input this line
****	quotes **^H**, your erase and kill
ESC	ends insertion, back to command
^?	interrupt, terminates insert
^D	backtab over *autoindent*
↑^D	kill *autoindent*, save for next
0^D	... but at margin next also
^V	quote non-printing character

Insert and replace

a	append after cursor
i	insert before cursor
A	append at end of line
I	insert before first non-blank
o	open line below
O	open above
r*x*	replace single char with *x*
R*text***ESC**	replace characters

Operators

Operators are followed by a cursor motion, and affect all text that would have been moved over. For example, since **w** moves over a word, **dw** deletes the word that would be moved over. Double the operator, e.g. **dd** to affect whole lines.

d	delete
c	change
y	yank lines to buffer
<	left shift
>	right shift
!	filter through command
=	indent for LISP

Miscellaneous Operations

C	change rest of line (**c$**)
D	delete rest of line (**d$**)
s	substitute chars (**cl**)
S	substitute lines (**cc**)
J	join lines
x	delete characters (**dl**)
X	... before cursor (**dh**)
Y	yank lines (**yy**)

Yank and Put

Put inserts the text most recently deleted or yanked. However, if a buffer is named, the text in that buffer is put instead.

p	put back text after cursor
P	put before cursor
"*x***p**	put from buffer *x*
"*x***y**	yank to buffer *x*
"*x***d**	delete into buffer *x*

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"*d***p**	retrieve *d*'th last delete

SEE ALSO

ex (1)

the *Vi* and *Ex* material in *ROS Text Editing Guide.*

CAVEATS AND BUGS

Software tabs using **^T** work only immediately after the *autoindent.*

Left and right shifts on intelligent terminals do not make use of insert and delete character operations in the terminal.

NAME

volmgr.test – test and query file system

SYNTAX

volmgr.test

DESCRIPTION

volmgr.test tests file system integrity, list file allocation and disc usage, and performs maintenance on the file system.

volmgr.test is an interactive program, using the standard input and output, with a set of self-explanatory commands and prompts.

NOTES

Volmgr.test should be used with caution; it can destroy the file system structure on the disc.

NAME

wait – await completion of process

SYNTAX

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

wall – write to all users

SYNTAX

/etc/wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by:

Broadcast Message from ...

It is used to warn all users, typically prior to shutting down the system.

The sender must be super-user to override any protections the users may have invoked (see *mesg*(1)).

FILES

/dev/tty*

SEE ALSO

mesg(1), write(1).

DIAGNOSTICS

"Cannot send to ..." when the open on a user's tty file fails.

NAME

wc – word count

SYNTAX

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

SYNTAX

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

whatis – describe what a command is

SYNTAX

whatis command ...

DESCRIPTION

Whatis looks up a given command and gives the header line from the manual section. You can then run the *man*(1) command to get more information. If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. Try 'whatis ed' and then you should do 'man 1 ed' to get the manual.

Whatis is actually just the **–f** option to the *man*(1) command.

FILES

/usr/lib/whatis Data base

SEE ALSO

man(1), catman(8)

NAME

whereis – locate source, binary, and or manual for program

SYNTAX

whereis [– **sbm**] [– **u**] [– **SBM** dir ... – **f**] name ...

DESCRIPTION

Whereis locates source/binary and manuals sections for specified files. The supplied names are first stripped of leading pathname components and any (single) trailing extension of the form ".ext", e.g. ".c". Prefixes of "s." resulting from use of source code control are also dealt with. *Whereis* then attempts to locate the desired program in a list of standard places. If any of the – **b**, – **s** or – **m** flags are given then *whereis* searches only for binaries, sources or manual sections respectively (or any two thereof). The – **u** flag may be used to search for unusual entries. A file is said to be unusual if it does not have one entry of each requested type. Thus "whereis -m -u *" asks for those files in the current directory which have no documentation.

Finally, the – **B** – **M** and – **S** flags may be used to change or otherwise limit the places where *whereis* searches. The – **f** file flags is used to terminate the last such directory list and signal the start of file names.

EXAMPLE

The following finds all the files in /usr/bin which are not documented in /usr/man/man1 with source in /usr/src/cmd:

```
cd /usr/ucb
whereis – u – M /usr/man/man1 – S /usr/src/cmd – f *
```

FILES

/usr/src/*
/usr/{doc,man}/*
/lib, /etc, /usr/{lib,bin,ucb,old,new,local}

BUGS

Since the program uses *chdir*(2) to run faster, pathnames given with the – **M** – **S** and – **B** must be full; i.e. they must begin with a "/".

NAME

who – who is on the system

SYNTAX

who [– **uTHlpdbrtasq**] [file]

who am i

who am I

DESCRIPTION

Who can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process-ID of the command interpreter (shell) for each current UNIX system 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** or **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** lists currently logged-in users only. The output *name* is the user's login name. The *line* is the name of the line as found in the **/dev** directory. *Time* is when the user logged in. *Activity* (WHICH IS NOT CURRENTLY IMPLEMENTED) gives the the number of hours and minutes since activity last occurred on that 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. 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 does not exist.

– **H** This option will print column headings above the regular output.

– **q** This is a quick *who*, displaying only the names and the number of users currently logged on. When this option is used, all other options are ignored.

– **p** 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

date(1), init(1), login(1), mesg(1), su(1), wait(2), inittab(4), utmp(4).

NAME

whoami – print effective current user id

SYNTAX

whoami

DESCRIPTION

Whoami prints who you are. It works even if you are su'd, while 'who am i' does not since it uses /etc/utmp.

FILES

/etc/passwd Name data base

SEE ALSO

who (1)

NAME

write – write to another user

SYNOPSIS

write user [line]

DESCRIPTION

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

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

to 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 superuser permissions, messages can be forced onto a write inhibited terminal.

If the character ! is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first *write* to another user, wait for them to *write* back before starting to send. Each person should end a message with a distinctive signal (i.e., **(o)** for "over") so that the other person knows when to reply. The signal **(oo)** (for "over and out") is suggested when conversation is to be terminated.

FILES

/etc/utmp to find user
/bin/sh to execute !

SEE ALSO

mail(1), mesg(1), nroff(1), pr(1), sh(1), who(1).

DIAGNOSTICS

"*user not logged in*" if the person you are trying to *write* to is not logged in.

NAME

xargs – construct argument list(s) and execute command

SYNTAX

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

EXAMPLES

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

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

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

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

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

```
1. ls |xargs – p – 1 ar r arch
2. ls |xargs – p – 1 |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

NAME

xstr - extract strings form C programs to implement shared strings

SYNTAX

xstr [-c] [-] [file]

DESCRIPTION

Xstr maintains a file *strings* into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

xstr -c *name* extracts the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file. The resulting C text is placed in the file *x.c*, to then be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffixes of existing strings do not cause changes to the data base.

After all components of a large program have been compiled, a file *xs.c* declaring the common *xstr* space can be created by a command of the form: **xstr** . This *xs.c* file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. **xstr** *name* creates files *x.c* and *xs.c* as before, without using or affecting any *strings* file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *Xstr* reads from its standard input when the argument '-' is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

```
cc -E name.c|xstr -c -
cc -c x.c
mv x.o name.o
```

Xstr does not touch the *strings* unless new items are added, thus *make* can avoid remaking *xs.o* unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array 'xstr'
/tmp/xs*	Temp file when 'xstr name' doesn't touch *strings*

SEE ALSO

mkstr(1)

BUGS

If a string is a suffix of another string in the data base, but the shorter string is seen first by *xstr* both strings will be placed in the date base, when just placing the longer one there will do.

NAME

yacc – yet another compiler-compiler

SYNTAX

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/yaccparparser	prototype for C programs

SEE ALSO

lex(1).

YACC– Yet Another Compiler Compiler in the *Programmers Guide to ROS.*

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

yes – be repetitively affirmative

SYNTAX

yes [expletive]

DESCRIPTION

Yes repeatedly outputs *expletive*. If the argument is omitted, "y" is output. Output is terminated by pressing the DEL key.

NAME

zero – clear floppy disc directory

SYNTAX

zero [-f] volumename

DESCRIPTION

Zero erases a floppy disc and initializes its directory for future use. The **volumename** of up to seven characters is written on the floppy directory. **-f** is used to format a new floppy before it can be used with Ridge system.

SEE ALSO

copyfloppy(1), dir(1)

NOTES

An error message and number is printed if the disc is not mounted, unreadable or unwriteable, or if the drive is not functioning correctly.

Ridge Computers
Corporate Headquarters

2451 Mission College Blvd.
Santa Clara, California 95054
Phone: (408) 986-8500
Telex: 176956